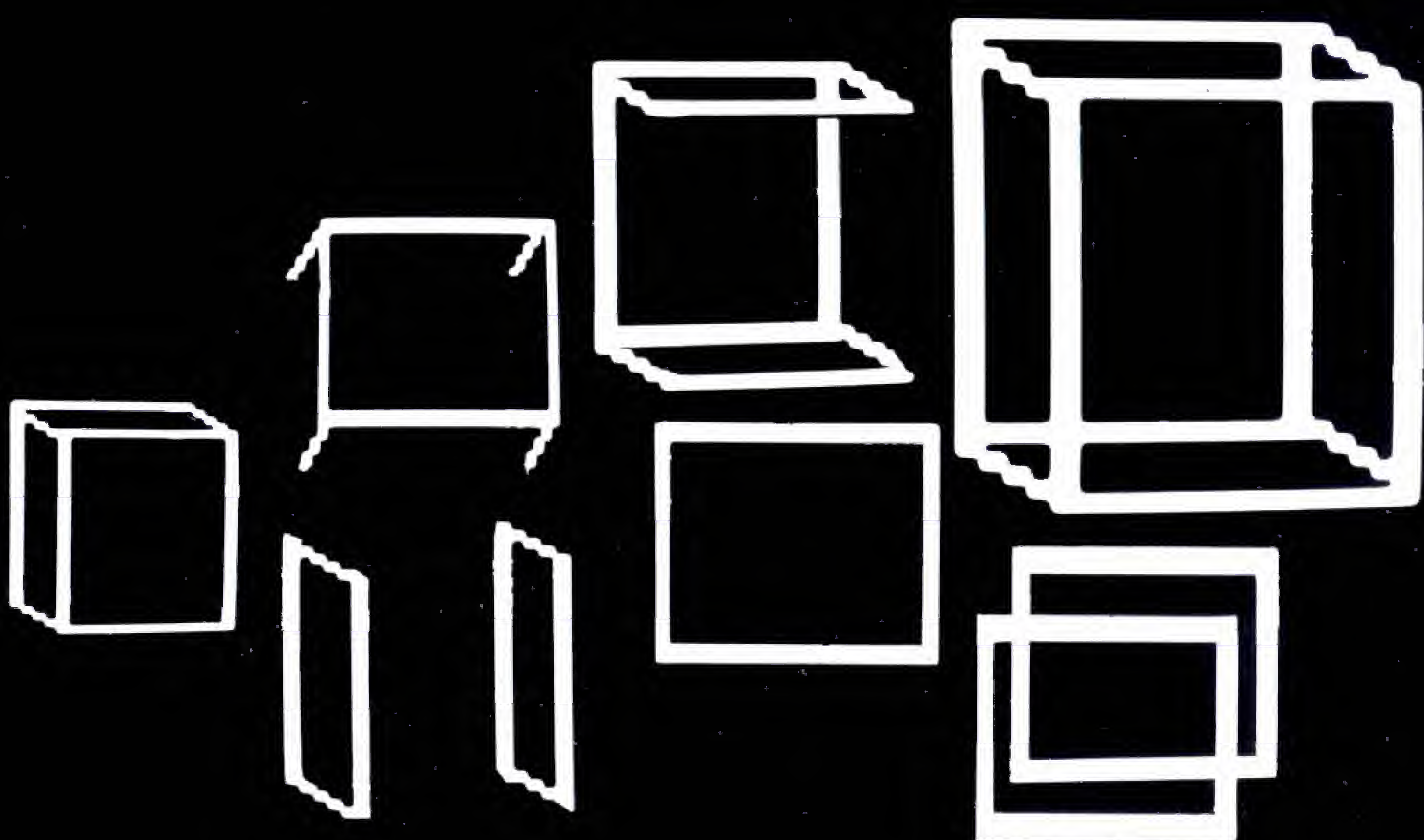
Computer Literacy Skills

GRAPHICS AND ANIMATION ON THE TRS-80 MODELS I, III, AND 4



CHRISTOPHER LAMPTON

GRAPHICS AND ANIMATION ON THE TRS-80

Models I, III, and 4

BY CHRISTOPHER LAMPTON

A COMPUTER LITERACY SKILLS BOOK

The words *computer graphics* and *animation* conjure up magical images of arcade games, bright colors, flashing lights, and wonderful shapes.

This book shows you how to create graphics using a TRS-80 Model I, III, or 4 microcomputer. Some knowledge of the BASIC language is needed, but you needn't be an expert programmer.

You will learn all the ins and outs of TRS-80 graphics, so that you can make the most of your machine's capabilities. Step-by-step techniques are blended with careful explanations of the internal workings and architecture of the TRS-80.

Numerous activities give you a chance to try nearly every technique presented. Suggested projects are given at the end of most chapters, and a graphics editing program will help you to channel your creativity more effectively.

So why depend on commercial programs to make your life more graphic and animated? Let Chris Lampton help you learn how to create your own games, designs, and charts.

FRANKLIN WATTS
387 PARK AVENUE SOUTH
NEW YORK, NEW YORK 10016

GRAPHICS AND ANIMATION ON THE TRS-80

Models I, III, and 4

By Christopher Lampton

A GROLIER COMPANY

A Computer Literacy Skills Book
FRANKLIN WATTS 1985
New York London Toronto Sydney

Diagrams by Vantage Art

Photographs courtesy of:
Daisy Taylor: p. 7 (top and bottom),
58 (top, left and right, and bottom);
Radio Shack/Tandy Corporation: pp. 30, 69, 89, 93, 96.

Computer graphics created by Sasha Petraske

Library of Congress Cataloging in Publication Data

Lampton, Christopher.
Graphics and animation on the TRS-80.

(A Computer literacy skills book)
Includes index.
Summary: Explains how to create and animate visual
displays on models I, III, and 4 of the TRS-80 computer.
1. TRS-80 computers—Programming—Juvenile literature.
2. Computer graphics—Juvenile literature. I. Title.
II. Series.
QA76.8.T18L36 1985 001.64'43 85-8978
ISBN 0-531-10059-6

Printed in the United States of America
6 5 4 3 2 1

CONTENTS

OTHER BOOKS BY
CHRISTOPHER LAMPTON

Advanced BASIC
BASIC *for Beginners*
Black Holes and Other Secrets of the Universe
COBOL *for Beginners*
Computer Languages
Dinosaurs and the Age of Reptiles
FORTH *for Beginners*
FORTRAN *for Beginners*
Fusion: The Eternal Flame
Graphics and Animation on the Commodore 64
Meteorology: An Introduction
PASCAL *for Beginners*
PILOT *for Beginners*
Planet Earth
Prehistoric Animals
Programming in BASIC
6502 *Assembly-Language Programming*
Space Sciences
The Sun
Z80 *Assembly-Language Programming*

GRAPHICS AND ANIMATION ON THE TRS-80

INTRODUCTION

THE PICTURE WINDOW

There is a world inside the computer where anything can happen—and nearly everything does, sooner or later. It is a world of bits and bytes, circuits and switches, but it is also a world of symbols and ideas, words and thoughts. As programmers, we control that world. We design the landscape and dictate the laws that govern the events that take place there.

Sometimes we want to open a window on that world—a picture window, as it were—so that we can see it with our own eyes. Computer graphics—visual images "drawn" on the video display of the computer—are the tool we use to build such a window. Graphics make the world inside the computer visible and real, whether it be the world of Pac-Man or of statistical formulas. They allow us to render scenes that no human eye has ever glimpsed—and to have a lot of fun in the process.

In the chapters that follow, we will learn how to create graphics on the video display of the TRS-80 Models I, III, and 4 microcomputers—and how to animate these images, make them move. We will create these images through the medium of TRS-80 Level II BASIC, the dialect of BASIC built into all of the computers just listed.

The TRS-80 features a low-resolution, black-and-white graphics display. The TRS-80 is not, to put it gently, the ideal graphics computer. But this should only inspire us to

greater heights as graphics programmers. The low-resolution graphics capabilities of the TRS-80 may seem unimpressive at first, but with a little imagination (and some advanced programming techniques), we can create some impressive displays.

TRS-80 Level II BASIC offers two methods of creating graphics: PRINT graphics and SET graphics. Each of these techniques has its strengths and its weaknesses, but both are important to the graphics programmer and will be discussed in this book.

All of our discussions will apply equally to the model I and model III computers. If you own a model 4, you will need to run it in model III mode. And if you have a model 4P, load the model III emulator from disk.

From here on, I'll assume you are familiar with the BASIC programming language and have some grasp of elementary programming techniques. (The BASIC course that comes with your TRS-80 computer should provide sufficient background.) If you are familiar with BASIC but not with the specific brand of BASIC found on the TRS-80, don't worry. You'll learn all features that are unique to this version, as well as obscure BASIC conventions you might not have encountered yet.

The key to graphics programming is the *pixel* (short for "pictorial element"), the basic unit from which all graphic images are created on the computer display. This is true whether the computer in question features high-resolution or low-resolution graphics, and whether the graphics are programmed in BASIC or machine language or any other programming language. The pixel usually appears on the computer's display as a tiny dot, or rectangle, of color. The difference between a high-resolution and a low-resolution computer is in the size of this pixel. A high-resolution computer has a small, dotlike pixel, a large number of which can fit onto the display. A low-resolution computer has a larger pixel, so fewer can fit onto the display.

Obviously, a small pixel is more desirable for creating graphics than a large pixel. A small pixel can be used to create shading and detail in a picture. A large pixel, on the other hand, will create blocky, chunky-looking images.

Unfortunately, the TRS-80 is a low-resolution comput-

er and supports only large pixels. No matter. Even within these limitations we can do a great deal with TRS-80 graphics.

On a black-and-white computer like the TRS-80, every pixel on the screen can have two states: black and white, on and off. Either the pixel is white, and therefore "on," or it is black, and therefore "off." (On a color computer, each pixel may take any of several colors.) We program graphic images on a computer by specifying which pixels we wish to turn on and which pixels we wish to turn off. However, we can use several different methods to specify which pixels should be on and off. Learning to program graphics is a matter of learning these methods.

Most computers have two primary methods of specifying pixels: *bit mapping* and *character graphics.* When using bit mapping, the programmer specifies every pixel individually, using a system of coordinates, as though the screen were a sheet of electronic graph paper. When using character graphics, on the other hand, the programmer is given a series of building blocks, each representing a predefined pattern of pixels. Graphic images are then put together from these building blocks, as a jigsaw puzzle is put together from precut pieces.

Each method has its own unique advantages. Bit mapping is the more flexible method, because it gives the programmer control over each individual pixel on the display. However, this flexibility is bought at a cost of more programming effort and slower program execution. Character graphics, on the other hand, are much easier to use, and take up far less of the computer's calculating time.

The method a programmer chooses will vary from situation to situation, from program to program (and sometimes computer to computer). Some programming problems lend themselves more to bit mapping, others to character graphics.

From here on, we will refer to bit mapping as SET graphics, because TRS-80 BASIC uses the SET command to create its own particular brand of bit mapping. We will refer to character graphics as PRINT graphics, because the PRINT command is primarily used to create this form of graphics in BASIC.

We will begin, in the next chapter, with SET graphics.

READY, SET, GO!

Only three commands in Level II BASIC are used purely for graphics: SET, RESET, and POINT. The SET command, which we'll discuss now, turns on a selected pixel on the TRS-80 display.

For a demonstration, prepare your computer for programming in BASIC. If you are running a cassette-based TRS-80, simply turn it on and press ENTER in response to the opening prompts. If you are using a disk system, turn it on, insert a formatted system disk, press RESET, type the date and time when requested, type BASIC, and press ENTER—and then press ENTER in response to the opening prompts.

To have the maximum amount of work space on your screen, press the key marked CLEAR. Everything on the screen will be erased, and a flashing cursor will appear in the upper left-hand corner of the display. Now, type the following:

```
SET(110,20)
```

and press ENTER. A white rectangle of light, a fraction of an inch on each side, will appear toward the right edge of the screen, about halfway between top and bottom. This pixel is the smallest piece of detail you will be able to include in a graphic image created on this computer. It may

seem rather large, when viewed in isolation like this, but think of it as a piece of plastic, or glass, from which you can construct elaborate mosaics.

How does the SET command work? How do we specify where on the TRS-80 screen we wish for a pixel to appear?

The two numbers in parentheses after the SET command are the *coordinates* of the pixel we want to turn on. The first is called the X coordinate, and represents the horizontal position of the pixel. The second, called the Y coordinate, represents the vertical position of the pixel.

THE PIXEL MATRIX

There are 6,144 pixels on the TRS-80 screen. When you enter BASIC, all of these pixels are off—that is, black. They remain off until you do something to turn them on. The chart on pages 8 and 9 shows where all of these pixels are. There are 128 pixels horizontally—that is, from left to right—and 48 pixels vertically—that is, from top to bottom. The horizontal pixels are numbered from 0 to 127, the vertical pixels from 0 to 47. Numbering begins at the upper left-hand corner of the screen. Therefore, to turn on the pixel in the upper left-hand corner, we would specify an X coordinate of 0 and a Y coordinate of 0, like this:

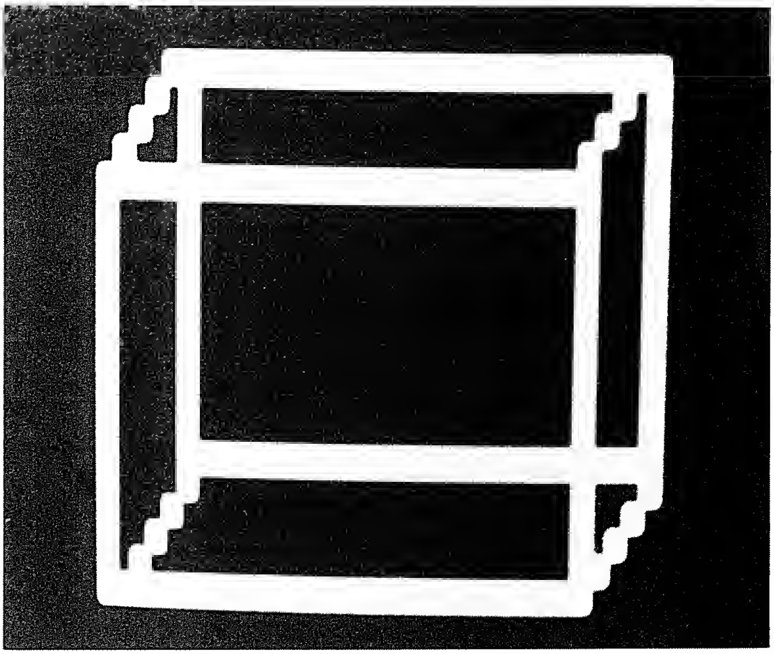
```
SET(0,0)
```

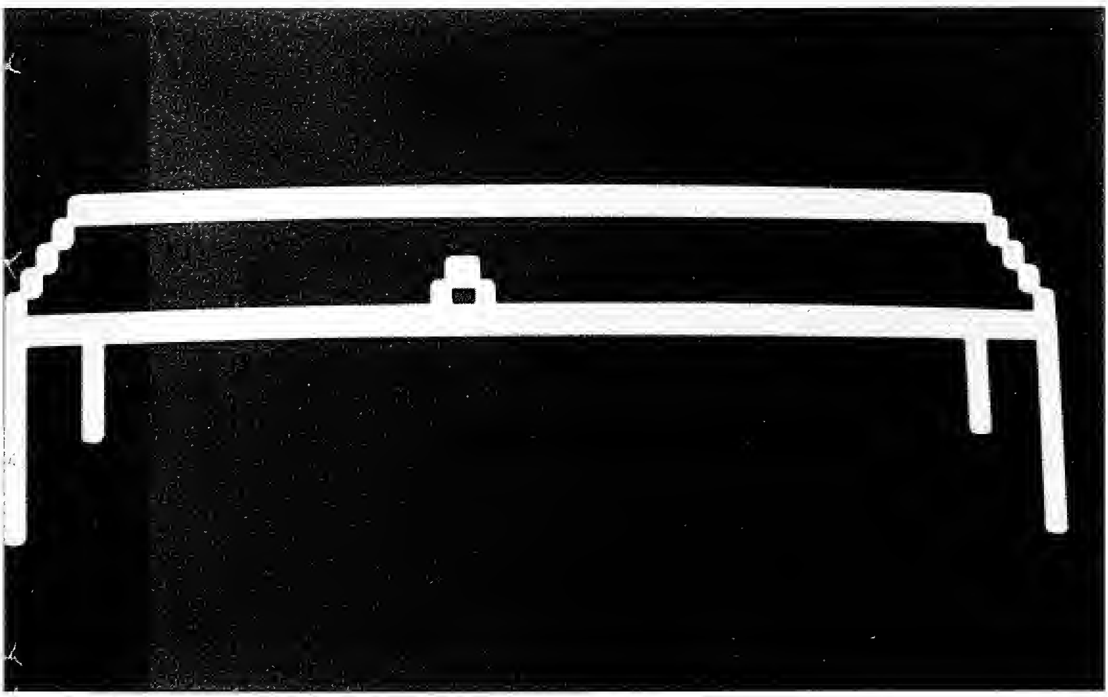
For every pixel that we move horizontally from this position, we add 1 to the first coordinate. For every pixel that we wish to move vertically from this position, we add 1 to the second coordinate. To turn on a pixel in the center of the screen, we would write:

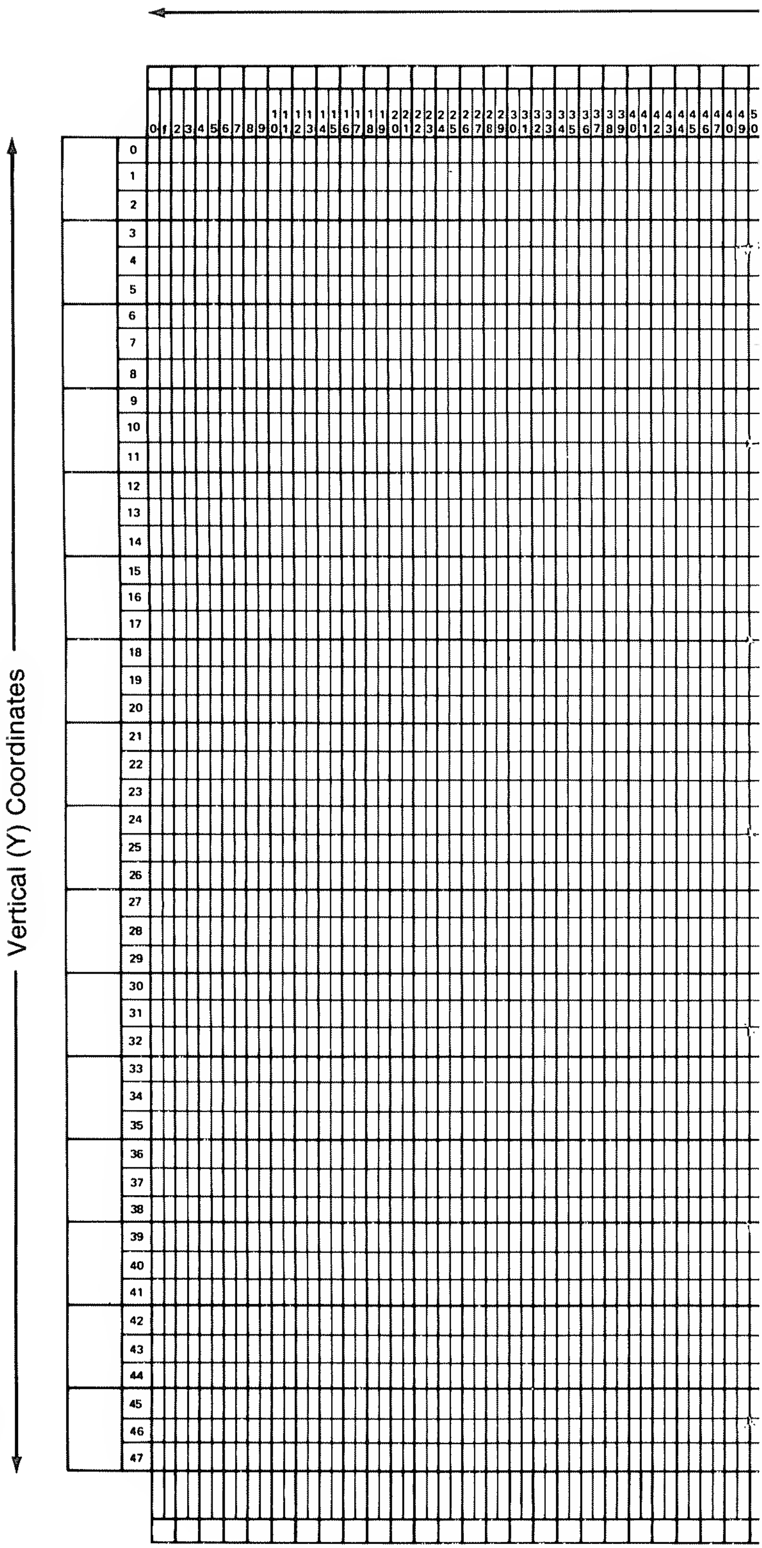
```
SET(64,24)
```

Naturally, these commands can be included within a BASIC program; we need not restrict ourselves to the

SET *graphics were used to construct these designs.*

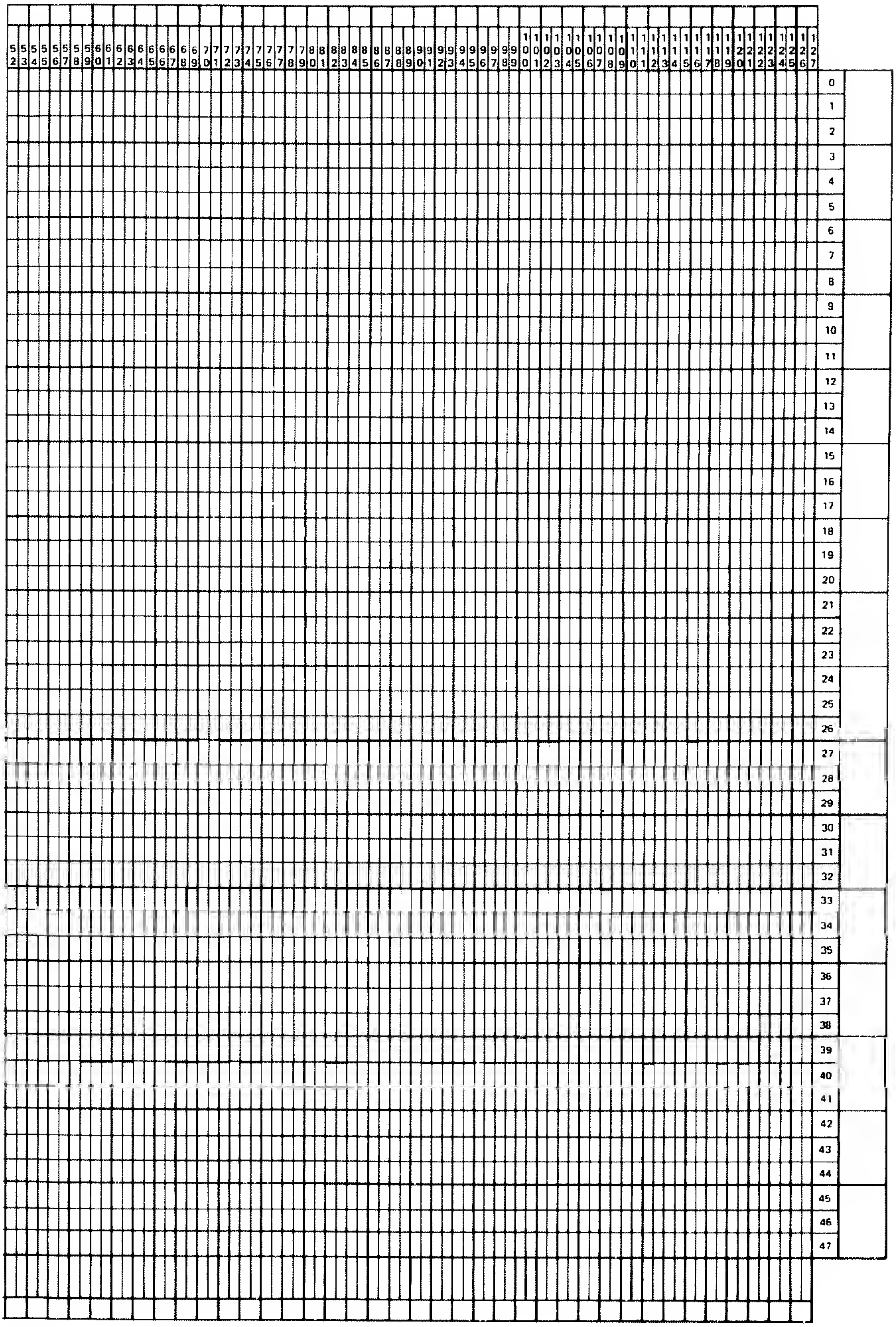






Screen Coordinates for SET, RESET, *and* POINT *Instructions.*

Horizontal (X) Coordinates



immediate mode. (In fact, we will not use them in the immediate mode again in this book.) Usually, we will want to turn on several pixels at a time, in a pattern. Complex designs are fairly easy to accomplish with the SET command, and can often be created with a few simple program loops.

Suppose, for instance, that we wish to draw a horizontal line across the screen using the SET command. How would we do this?

Well, one characteristic of a horizontal line made up of a single row of pixels is that every pixel in the line has the same vertical coordinate but a different horizontal coordinate. We can SET the pixels on a line by placing the SET command in a program loop and continually incrementing the X coordinate, leaving the Y coordinate unchanged, like this:

```
1 REM *** HORIZONTAL LINE
5 CLS
10 FOR X = 0 TO 127
20 SET(X,24)
30 NEXT X
40 GOTO 40
```

Type this program and RUN it. It will draw a line across the entire horizontal width of the screen, cutting the display in half. We have, you may note, used the variables X and Y in this program to represent the X and Y coordinates of the pixel. Although we have chosen these variable names because they clarify the purpose of the variables, you need not restrict yourself to these variable names in the SET command.

(For those of you not already familiar with the TRS-80 dialect of BASIC, note that the command CLS in the first line clears the video display, just as though you had pressed the CLEAR key.)

What we have done in this program is to establish a FOR-NEXT loop that steps variable X through all possible whole numbers from 0 to 127—that is, all the possible values that the X coordinate of a SET command may take. On each pass through the loop, the SET command in line 20 turns on a pixel at a different X coordinate. However, the Y

coordinate remains fixed at 24, so all of these pixels are in the vertical center of the screen—in a straight line. The final line of the program, line 40, creates an infinite loop that holds the graphic image on the screen long enough for us to admire it. To terminate the program, press the key marked BREAK.

To create a vertical line, we increment the value of the Y coordinate while letting the X coordinate remain the same, like this:

```
1 REM *** VERTICAL LINE
5 CLS
10 FOR Y = 0 TO 47
20 SET(64,Y)
30 NEXT Y
40 GOTO 40
```

This program, when RUN, will draw a line from the top of the screen to the bottom, in the same manner that the last program drew a line from one side to the other. Compare this program with the last, and notice how few alterations were necessary to make this change.

To draw a diagonal line, we must simultaneously increment the values of the X and Y coordinates. This is trickier, since the FOR-NEXT loop will increment only one of these values at a time. Thus, we must increment the second one independently. Here is one way to do this:

```
1 REM *** DIAGONAL LINE
5 CLS : X = 0
10 FOR Y = 0 TO 47
20 SET(X,Y)
30 X = Y + 1
40 NEXT Y
50 GOTO 50
```

Now the value of the X coordinate is increased by the instruction in line 30, which adds 1 to the value of variable X on each pass through the loop. The value of Y, as before, is incremented by the FOR-NEXT loop. When RUN, this program will draw a line beginning at the upper left-hand corner of the screen and terminating at the bottom, about

one-third of the way across. By changing the values at which we begin and end the incrementing of each variable, we can place the line at any point on the screen we wish. Be careful that neither variable exceeds the allowed range of coordinate values, or the program will be interrupted by an error message. For instance, if we had placed X in the FOR-NEXT loop instead of Y, and had stepped X through the full range of values from 0 to 127, we would have received an error message as soon as Y had exceeded 47.

It is possible to draw diagonal lines that slope at any angle (within the resolution of the TRS-80 screen). To change the angle, we must change the amount by which we increment the X or the Y coordinate. Ideally, in order to maintain an even line with no gaps, one of these coordinates should be incremented by 1 (generally through a FOR-NEXT loop), the other by a fractional value smaller than 1. (Note that any fractional value in a screen coordinate will be ignored by the SET command, though the value of the variable, if any, that represents that coordinate will be unchanged.)

For example, the following variation on the last program changes the angle of the line by incrementing the value of X by 0.5 rather than 1. The Y coordinate is still incremented by a FOR-NEXT loop.

```
1 REM *** DIAGONAL LINE
5 CLS : X = 0
10 FOR Y = 0 TO 47
20 SET(X,Y)
30 X = X + .5
40 NEXT Y
50 GOTO 50
```

For certain graphics applications, it is useful to have a subroutine that will draw a line between any two specified points on the screen. (Some versions of BASIC feature a command, such as LINE or DRAW, that performs this task, but TRS-80 BASIC does not.) Such a subroutine would need to decide whether to increment the X coordinate or the Y coordinate by a FOR-NEXT loop and by what value to increment the other coordinate. The subroutine would then set the actual pixels in the line.

[13]

Here is a subroutine that will perform all of these tasks on the TRS-80:

```
10 REM *** DRAW A LINE BETWEEN 2 POINTS
20 REM This subroutine draws a line between
30 REM coordinates (X1,Y1) and (X2,Y2).
40 REM
10000 DX = X2 - X1 : DY = Y2 - Y1
10010 IF ABS(DX) < ABS(DY) THEN 10080
10020 YI = DY/ABS(DX)
10030 FOR X = X1 TO X2 STEP SGN(DX)
10040 SET(X,Y1)
10050 Y1 = Y1 + YI
10060 NEXT
10070 RETURN
10080 XI = DX/ABS(DY)
10090 FOR Y = Y1 TO Y2 STEP SGN(DY)
10100 SET(X1,Y)
10110 X1 = X1 + XI
10120 NEXT
10130 RETURN
```

(Lines 10 through 40 may be safely removed from this routine before you use it in a program.)

No detailed explanation of this routine will be given here, but you can use it in your own programs exactly as it is given. When calling this subroutine, place the X,Y coordinates of the start of the line in variables X1 and Y1, and the coordinates of the end of the line in variables X2 and Y2. For instance, to draw a line between (100,30) and (2,41), you would use a calling routine like this:

```
10 CLS
20 X1 = 100 : Y1 = 30 : X2 = 2 : Y2 = 41
30 GOSUB 10000
40 GOTO 40
```

This subroutine has many practical (and impractical) uses. Chapter Seven shows how to use it in a program that draws graphs based on numerical values. Here, however, is a relatively frivolous calling routine that uses the subroutine to scribble random lines on the screen:

```
10 CLS
20 X1 = 64 : Y1 = 24
30 X2 = RND(127) : Y2 = RND(47)
40 GOSUB 10000
50 X1 = X2 : Y1 = Y2
60 GOTO 30
```

Hit BREAK when you get tired of watching the random drawing (or when the screen gets too cluttered for you to see the new lines being drawn).

GEOMETRIC SHAPES

You may want to use the SET command to draw simple geometric shapes on the computer screen. Although you could use the sophisticated line-drawing routine above for such a purpose, the routine really provides more drawing power than we may need. A simple FOR-NEXT loop—or group of several loops—will usually accomplish the task.

The following program, for instance, draws a rectangle in the center of the display, using two sequential FOR-NEXT loops:

```
1 REM *** RECTANGLE
5 CLS
10 FOR X = 58 TO 70
20 SET(X,20) : SET(X,28)
30 NEXT X
40 FOR Y = 20 TO 28
50 SET (58,Y) : SET (70,Y)
60 NEXT Y
70 GOTO 70
```

One of the loops draws the horizontal lines; one draws the vertical lines. Notice that each loop draws two lines simultaneously, for a total of four lines. By enlarging our rectangle to its absolute limit, we can draw a decorative border around the entire video display, like this:

```
1 REM *** DECORATIVE BORDER
5 CLS
10 FOR X = 0 TO 127
```

```
20 SET(X,0) : SET(X,47)
30 NEXT X
40 FOR Y = 0 TO 47
50 SET (0,Y) : SET(127,Y)
60 NEXT Y
70 GOTO 70
```

Once again, by adjusting the starting and ending values of the loops, we can draw the rectangle in any size we wish, with any ratio of height to width, within reasonable limits.

SET graphics are a good way to decorate an otherwise dull program display. You can put borders around the display, as just demonstrated, or even divide the screen into several rectangular "windows," each containing a unique portion of the information being displayed.

An even more practical use of SET graphics is the creation of bar charts, pictorial representations of large masses of numbers. A computerized sales report, for instance, might be accompanied by an on-screen chart using pixel-generated bars to compare the sales of two different items, or the total sales for two different years.

Here is a short program that will draw such a bar on the display of the TRS-80:

```
1 REM *** RECTANGULAR BAR
10 CLS
20 FOR Y = 47 TO 20 STEP -1
30 FOR X = 60 TO 70
40 SET(X,Y)
50 NEXT X
60 NEXT Y
70 GOTO 70
```

The bar is drawn from the bottom of the screen upward by a pair of nested FOR-NEXT loops. The vertical dimension of the bar is drawn by the outer (Y) loop; the horizontal dimension is drawn by the inner (X) loop. The bar is 28 pixels tall, from Y coordinate 47 at the bottom to Y coordinate 20 at the top.

Few bar charts feature a single bar, however; at least two bars are necessary, so that the size of the bars can be

visually compared. Here is a variation on the above program that draws a pair of bars on the display:

```
5 REM *** TWO RECTANGULAR BARS
10 CLS
20 FOR I = 0 TO 1
30 FOR Y = 47 TO 20 STEP -1
40 FOR X = 35 + I * 45 TO 35 + I * 45 + 10
50 SET(X,Y)
60 NEXT X
70 NEXT Y
80 NEXT I
90 GOTO 90
```

The bars themselves are still drawn by a pair of nested FOR-NEXT loops. However, we have now nested those two loops inside a third loop, which will execute twice, once for each bar to be drawn. The index variable of this new loop is I, and we have used it in line 40 to calculate the X coordinates of each bar. The horizontal width of the bar extends from X coordinate 35 + I * 45 to X coordinate 35 + I * 45 + 10. The first time the loop executes, I will be equal to 0. Thus, the first bar will extend horizontally from X coordinate 35 to X coordinate 45, a width of eleven pixels. The second time the loop executes, I will be equal to 1. Thus, the second bar will extend horizontally from X coordinate 80 to X coordinate 90, also a width of eleven pixels.

Both of these bars, however, will have the same vertical dimensions; that is, they will be the same height. For a bar chart to be useful, we must be able to draw bars of varying heights, to visually represent varying quantities. Here is a variation on the above program that prompts the user to type in a pair of numbers from the keyboard, then draws two bars visually comparing these numbers. The first bar will have a height in pixels equal to the first number typed; the second bar will have a height in pixels equal to the second number typed:

```
1 REM *** SIMPLE BAR CHART
10 CLS
20 INPUT "TYPE TWO NUMBERS BETWEEN 1 AND 47";
B(0), B(1)
```

```
30 IF B(0) < 1 OR B(0) > 47 OR B(1) < 1 OR B(1) > 47
THEN 20
40 FOR I =0 TO 1
50 FOR Y = 47 TO 47-B(1) STEP -1
60 FOR X = 35 + I * 45 TO 35 + I * 45 + 10
70 SET(X,Y)
80 NEXT X
90 NEXT Y
100 NEXT I
110 GOTO 110
```

The INPUT statement in line 20 places the two typed values in variables B(0) and B(1), respectively. Line 30 checks to make sure these values are in the allowed range; if they are not, the user is prompted for two more numbers. The FOR-NEXT loop in line 50 has been rewritten to step the value of coordinate Y through a range of pixels equal to the value of variable B(1). On the first pass through the outer loop, when the first bar is drawn, this will represent variable B(0), which is equal to the first number typed. On the second pass through the outer loop, when the second bar is drawn, this will represent variable B(1), which is equal to the second number typed. Thus, the first bar will be drawn to the height specified by the first number typed, and the second bar will be drawn to the height specified by the second number typed.

Type the program and RUN it. Experiment with various values. In Chapter Seven, we will see how this program can be dressed up and turned into a full-fledged business chart program, with several bells and whistles.

DRAWING WITH THE KEYBOARD

Another way that we can put SET graphics to use is in the creation of a program that allows us to draw pictures on the computer screen by pressing certain keys on the keyboard. Such a program can be useful in the creation of graphic designs; it can also be a lot of fun to play with.

In order to draw from the keyboard, we need a way to interact with the program while it is in progress, in order to direct the creation of the drawing. The INPUT command is

not appropriate for this purpose, because it will cause the program to stop executing unless you press the ENTER key. Worse, a question mark will be printed on the screen, destroying your graphics display. Therefore, we need a way to detect the pressing of a single key without otherwise interrupting the flow of the program.

In TRS-80 BASIC, this is done with the INKEY$ function. Like a string variable, INKEY$ is always equal to a string of characters and thus can be used in a program in much the same way we normally use string variables (that is, variables that are set equal to strings of characters). Unlike an ordinary string variable, however, INKEY$ cannot be assigned string values in an assignment statement. Rather, Level II BASIC automatically assigns a string to INKEY$ that represents the last key pressed on the keyboard.

If we press the A key, for instance, INKEY$ will be set equal to the single character string A. If we then press the 5 key, the value of INKEY$ will become 5. On the other hand, if no key has been pressed INKEY$ will be equal to the *null string,* or *empty string,* the string that contains no characters at all.

The null string is represented within a BASIC program by a pair of quotation marks (""). Note that there is no blank space between these marks, since the blank space is an actual character that can be typed at the computer keyboard by pressing the space bar.

The following short demonstration program will read the TRS-80 keyboard with INKEY$ and echo each key to the screen as it is typed:

```
10 A$ = INKEY$
20 IF A$ = "" THEN 10
30 PRINT A$;
40 GOTO 10
```

Line 10 sets string variable A$ equal to the current value of INKEY$. Line 20 checks for a null string value, which would indicate that no key has been pressed yet. If found, the program will loop back to look again. If a key *has* been pressed—that is, if the value of A$ is not null—the string represented by that key is printed on the screen by line 30

and the GOTO statement in line 40 leaps back to 10 to repeat the process.

CONTROL CHARACTERS

Not every key on the keyboard produces what we think of as a "character." Some keys cause actions to take place on the video display. For example, press the key marked ENTER while running the above program. No character will be printed on the display. Nonetheless, something does happen when this key is pressed. Ordinarily, this program will print each character in the position immediately following the previous character. After pressing ENTER, however, you will find that subsequent characters are printed on the following line of the display, beginning at the left-hand margin.

The ENTER key produces a *carriage return;* that is, it moves the position at which new characters are printed down one line and back to the left margin, in the same way the carriage return key on a typewriter moves the printhead to a new line and back to the margin. Although no visible character is printed when we press ENTER, we can say that it produces a *control character,* an imaginary character that causes an action to take place on the screen. The INKEY$ function will be set equal to this control character in the same way that it becomes equal to any other character on the keyboard.

The four arrow keys on the TRS-80 keyboard—the right arrow, left arrow, up arrow, and down arrow—also produce control characters. In normal BASIC programming mode, the left arrow backspaces and deletes the previously typed character, the right arrow tabs forward eight spaces, the down arrow produces a line feed (a control character similar to a carriage return), and so forth. For our sketch program, it would be useful if we could detect the pressing of these arrow keys, since they could be used to direct the drawing of an image on the computer's screen. Here is a short program that demonstrates how INKEY$ can be used to detect the pressing of the arrow keys:

```
10 A$ = INKEY$
20 IF A$ = "" THEN 10
```

```
30 IF A$ = CHR$(8) THEN PRINT "YOU PRESSED THE LEFT
ARROW"
50 IF A$ = CHR$(9) THEN PRINT "YOU PRESSED THE RIGHT
ARROW"
60 IF A$ = CHR$(10) THEN PRINT "YOU PRESSED THE DOWN
ARROW"
70 IF A$ = CHR$(91) THEN PRINT "YOU PRESSED THE UP
ARROW"
80 GOTO 10
```

RUN this program and press various keys on the TRS-80 keyboard. Nothing should happen, unless one of the arrow keys is pressed, in which event the program will identify the key that is pressed and continue waiting for you to press a key.

This program identifies the pressing of an arrow key by the ASCII code number that it produces. ASCII (a term you may have encountered before now) is short for American Standard Code for Information Interchange. ASCII is a coding system that assigns numbers to characters, such as letters of the alphabet and control characters. Every character that we can produce on the TRS-80 keyboard, including control characters such as carriage returns, backspaces, and so on, has a corresponding ASCII code.

THE CHR$ FUNCTION

The BASIC function CHR$ converts a number (always in the range 0 to 255) into the ASCII character it represents. CHR$(65), for instance, is the letter "A", because 65 is the code number for "A". CHR$(13) is a carriage return. And so on. When we press the keys that produce these characters, the appropriate ASCII code number is stored at a special memory location within the TRS-80 called the *key latch*. It is the character value in the key latch that is assigned to the INKEY$ function.

We can use the CHR$ function to detect whether INKEY$ has been set equal to one of the characters produced by the arrow keys. CHR$(8) is the control character produced by the left arrow key, CHR$(9) is the control character produced by the right arrow key, CHR$(10) is the control character produced by the down arrow key, and CHR$(91) is the control character produced by the up arrow key.

These figures are important enough for us to incorporate them into a little chart:

KEY	CHR$ VALUE
Left Arrow	8
Right Arrow	9
Down Arrow	10
Up Arrow	91

With this information, it is possible to write our first, crude sketch program, like this:

```
1 REM *** SKETCH
5 CLS
10 X = 64 : Y = 24
20 SET(X,Y)
30 A$ = INKEY$
40 IF A$ = CHR$(8) THEN X = X-1
50 IF A$ = CHR$(9) THEN X = X + 1
60 IF A$ = CHR$(10) THEN Y = Y + 1
70 IF A$ = CHR$(91) THEN Y = Y-1
80 GOTO 20
```

RUN this program. The screen will clear, and a single pixel will appear in the center of the display. Tap the up arrow key. Another pixel will appear directly above the first one. Tap the up arrow key a second time. Yet another pixel will appear, immediately above the first two. Tap the up arrow key yet again. A line of pixels has begun to form.

Now tap the left arrow key. The next pixel will appear to the left of the line. Tap the left arrow key a few more times. A new line will begin to extend at right angles to the first.

Tap any of the arrow keys and the line will extend in the direction pointed to by the arrow. By manipulating these lines, you can actually draw a picture on the video display.

As you can see, this is quite a simple program, and it has its limitations. For instance, if you extend a line too far in any one direction, you will run right off the edge of the screen—and the program will be interrupted by an error message (obliterating the beautiful picture that you have

drawn). Later, you will learn how to trap out such errors, along with some useful ways to extend the program, turning it into a valuable graphics utility.

First, however, let's look at a slightly different aspect of TRS-80 SET graphics: animation.

Suggested Projects

1. Write a program that will draw a line from the upper left-hand corner of the display to the lower right-hand corner.

2. Write a program that turns the display of the TRS-80 completely white.

3. Expand the bar chart program in this chapter. Allow the user to input the number of bars desired (within a reasonable range) and the length of each bar; then have the program draw the chart on the display. (For one solution to this problem, see Chapter Seven.)

4. ADVANCED PROJECT: Write a program that draws a curve or a circle on the display. (Don't become frustrated if you can't make this program work. It's a difficult task, and a knowledge of trigonometry is helpful.)

A MOVING EXPERIENCE

Although most of the animation in this book will be performed using other techniques, it is possible to create a crude sort of animation using pixel graphics. Type this program:

```
10 CLS
20 FOR X = 0 TO 127
30 SET(X,24) : RESET(X,24)
40 NEXT X
50 GOTO 50
```

The sharp-eyed among you will notice that this is identical to our first line-drawing program, with the addition of a single statement: RESET(X,24).

The RESET command, as the name implies, is the reverse of the SET command. It turns off the pixel at a specified coordinate position on the display. The coordinate system for the RESET command is identical to that for the SET command. Thus, the RESET statement in the above program simply turns off the pixel turned on by the SET statement.

What good does this do us? RUN the program. Instead of seeing a line drawn across the video display, we see a lone (rather flickery) pixel racing across the screen, from

the left to the right, looking somewhat like a projectile fired by a spaceship in an arcade game such as Defender. Although the animation created here is crude, it is effective. The pixel actually seems to move.

How is this effect accomplished? As in a motion picture or animated cartoon, we are essentially creating a sequence of *frames* on the computer screen and displaying each in turn. In the first frame, the pixel is on the far left side of the display, turned on by the SET command in line 30. Then it is turned off by the RESET command, effectively blanking the frame. The loop repeats, creating a second frame with the pixel in the next horizontal position, and so forth.

Thus the pixel is animated. As unsophisticated as this animation may seem, it demonstrates the procedure that underlies all of the graphic animation that we will study in this book: draw an image on the screen, erase the image, and redraw it in a new position.

GETTING RID OF FLICKER

This program could stand improvement. Because the pixel is turned off almost immediately after it is turned on, and because it remains dark for the greater part of the loop, it actually spends more time off than on, which makes it seem dim and flickery, difficult to see. We can reduce this flicker effect by increasing the amount of time between the execution of the SET and RESET instructions.

Consider this version of the program:

```
10 CLS
20 FOR X = 0 TO 127
25 IF X = 0 THEN 35
30 RESET(X-1,24)
35 SET(X,24)
40 NEXT X
50 GOTO 50
```

We have now reversed the positions of the SET and RESET commands. RESET turns off the *previous* pixel (at horizontal coordinate X − 1) immediately before the current pixel is turned on, allowing the pixel to remain visible for the maximum amount of time. The IF statement in line 25 causes

this RESET statement to be skipped during the first execution of the loop, when there is no previous pixel to turn off.

RUN this version of the program. The moving pixel is now clear and distinct, with only a slight flicker.

To change the direction in which the pixel moves, we need only change the way we increment the X,Y coordinates, just as we changed the direction of the lines we drew in the previous chapter. In fact, any of the line-drawing programs in Chapter One can be changed into an animation program, simply by adding a statement that erases the previous pixel before or after a new pixel is drawn.

Our sketch program, for instance, can be changed into a program that allows us to move a pixel about the screen by pressing the arrow keys. There is no practical use for such a program—no picture is created on the display—but it illustrates techniques that could be used in an arcade game, where the motion of a spaceship or other on-screen object can be controlled by pressing keys on the keyboard.

Here is such a version of the sketch program:

```
10 CLS
20 X = 64 : Y = 24
30 SET(X,Y)
40 LX = X : LY = Y
50 A$ = INKEY$
60 IF A$ = CHR$(8) THEN X = X -1
70 IF A$ = CHR$(9) THEN X = X + 1
80 IF A$ = CHR$(10) THEN Y = Y + 1
90 IF A$ = CHR$(91) THEN Y = Y -1
100 RESET(LX,LY)
110 GOTO 30
```

Now, when you RUN the program, a flickering pixel will appear in the middle of the screen. When an arrow key is pressed, the pixel will move in the direction indicated by the arrow.

As before, the variables X and Y contain the X,Y coordinates of the pixel. The two variables LX and LY contain the *previous* (or "last") values of X and Y. This allows us to reset the previous pixel image in line 100, after the values of X and Y have been altered by lines 60 through 90.

STOP THAT ERROR

This program suffers from the same problem inherent to the original sketch program: it is possible to move the pixel off the edge of the screen, causing an error message. Before going on, we must find a way to prevent this from happening, so that the problem will not afflict future programs presented in this book.

The easiest way to forestall such an occurrence is to have the program test the next value of the X and Y coordinates, before we actually change these values, to see if either will be off the edge of the screen. Several methods could be used to do this; the one we will explore now involves creating a pair of "direction" variables—variables that contain values representing the next direction, left or right, up or down, in which the X and Y coordinates will be changed.

We will call these variables DX and DY (for "direction of X" and "direction of Y"). Each can assume one of three different values: −I, 0, or 1. A value of 0 means that the coordinate, X or Y, will not be changed during the current loop; a value of −1 means that the number 1 will be subtracted from the coordinate (moving the X coordinate to the left or the Y coordinate up); and a value of 1 means that the number 1 will be added to the coordinate (moving the X coordinate to the right or the Y coordinate down).

We then test the values of X + DX and Y + DY to see if they are off the edge of the screen. If they are, we don't move the pixel during that loop. If they aren't, then we add DX to X and DY to Y. Here's the program itself:

```
10 CLS
20 X = 64 : Y = 24
30 DX = 0 : DY = 0
40 SET(X,Y)
50 LX = X : LY = Y
60 A$ = INKEY$
70 IF A$ = CHR$(8) THEN DX = −1
80 IF A$ = CHR$(9) THEN DX = 1
90 IF A$ = CHR$(10) THEN DY = 1
100 IF A$ = CHR$(91) THEN DY = −1
110 IF X + DX < 0 OR X + DX > 127 OR Y + DY < 0 OR
Y + DY > 47 THEN 30
```

```
120 X = X + DX : Y = Y + DY
130 RESET(LX,LY)
140 GOTO 30
```

REPEATING KEYS

Another way to improve this program is to give it a repeat-key capability. You'll notice that the pixel only moves once every time you press an arrow key. You must raise your finger from the key and press it again to repeat the movement. It would be much nicer if the pixel continued moving as long as you held down the key, the way a spaceship in an arcade game might. Alas, this cannot be done with the INKEY$ function. Every time we read the value of INKEY$ in a program (as we do in line 60 of this program), INKEY$ is automatically set equal to the null string again until another key is pressed (or until the same key is pressed again). Thus, the pixel will move one position every time we press an arrow key; then it will stop.

The best way around this problem is to avoid INKEY$ altogether and use the *keyboard matrix*. The keyboard matrix is a portion of the TRS-80's memory that contains information about which keys are currently being pressed. (For those of you not familiar with the terminology, *memory* is a series of electronic circuits inside the computer, each of which holds information in the form of a number between 0 and 255. All information processed by the computer, including programs, is contained in these circuits. Each of these circuits is assigned an identifying number called an *address*. Certain memory addresses are used for special purposes, such as the keyboard matrix. We describe some of these purposes below. You should realize, however, that the memory addresses we describe here are used for these purposes *only* on the TRS-80 models I, III, and 4. Other machines will use their memory differently.)

We can use the BASIC function PEEK to look at the keyboard matrix. The PEEK function is always equal to the value stored at a specified memory address. For instance, PEEK(12345) is equal to the value stored at memory address 12345.

The keyboard matrix resides in the TRS-80's memory between addresses 14337 and 14464. For the moment, we

are interested only in address 14400. By PEEKing this location, we can get information about the current status of the space bar, ENTER, CLEAR, BREAK, up arrow, down arrow, left arrow, and right arrow keys.

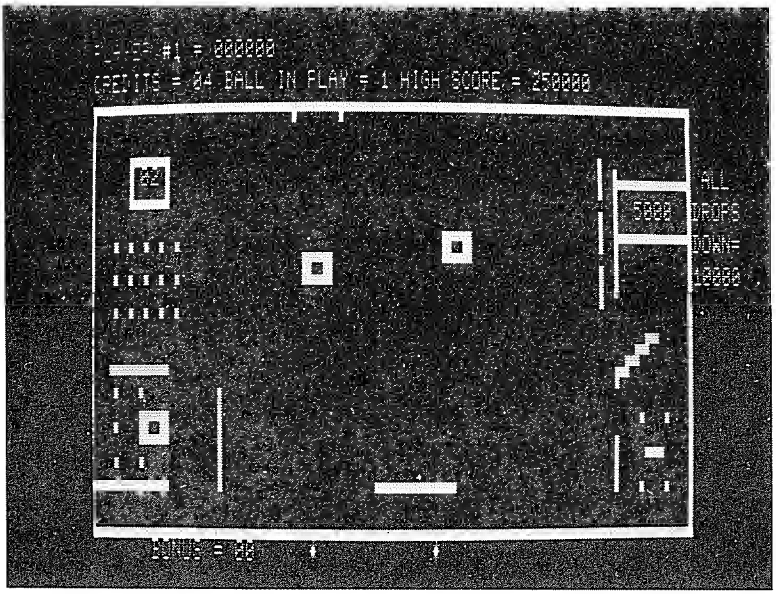
The value of PEEK(14400) tells us which of these keys is being pressed, according to this chart:

KEY	VALUE OF PEEK(14400)
ENTER	1
CLEAR	2
BREAK	4
Up Arrow	8
Down Arrow	16
Left Arrow	32
Right Arrow	64
Space Bar	128

We can easily change our program so that it reads the keyboard by PEEKing the keyboard matrix, like this:

```
10 CLS
20 X = 64 : Y = 24
30 DX = 0 : DY = 0
40 SET(X,Y)
50 LX = X : LY = Y
60 A = PEEK(14400)
70 IF A = 8 THEN DY = -1
80 IF A = 16 THEN DY = 1
90 IF A = 32 THEN DX = -1
100 IF A = 64 THEN DX = 1
110 IF X + DX < 0 OR X + DX > 127 OR Y + DY < 0 OR
Y + DY > 47 THEN 30
120 X = X + DX : Y = Y + DY
130 RESET(LX,LY)
140 GOTO 30
```

Now the pixel will move continuously across the screen as you hold down an arrow key. If you remove lines 50 and 130, which are concerned with resetting the previous image of the pixel, you can turn this into a more advanced version of the sketch program described in the last chapter.



Pong-type game

FUN AND GAMES

We now know enough about the graphics and animation capabilities of the TRS-80 to create a complete arcadelike game program, somewhat along the lines of the inexpensive Pong games that used to be sold as attachments for home televisions. In the following program, you are given control of a graphic "paddle" on the left-hand side of the display, with which you must hit a moving pixel that bounces randomly off the white "walls" that border the remainder of the screen. The paddle is controlled with the two arrow keys on the left-hand side of the keyboard. You will gain a point every time you hit the ball successfully. Miss the ball and the game will be terminated.

```
10 CLS
20 SC = 0 : REM INITIALIZE SCORE
30 FOR X = 9 TO 110 : REM DRAW TOP AND BOTTOM
BORDERS
40 SET(X,3) : SET(X,47)
50 NEXT X
```

```
60 FOR Y = 3 TO 47 : REM DRAW RIGHT BORDER
70 SET(110,Y)
80 NEXT Y
90 FOR Y = 22 TO 26 : REM DRAW PADDLE
100 SET(9,Y)
110 NEXT Y
120 PRINT@4, "SCORE = 0";
130 PY = 22 : REM Y COORDINATE FOR TOP OF PADDLE
140 X = 15 : Y = 20 : REM STARTING X,Y COORDINATES FOR
BALL
150 LX = X : LY = Y
160 DY = 1 : REM Y DIRECTION
170 DX = 1 : REM X DIRECTION
180 A = PEEK(14400) : REM SCAN KEYBOARD MATRIX
190 IF A = 16 AND PY < 42 THEN RESET(9,PY) : SET(9,PY + 5)
: PY = PY + 1
200 IF A = 8 AND PY > 4 THEN RESET(9,PY + 4) : SET(9,PY
-1) : PY = PY -1
210 RESET(LX,LY) : REM ERASE OLD PIXEL
220 SET(X,Y) : REM SET NEW PIXEL
230 LX = X : LY = Y
240 IF POINT(X+DX,Y) =-1 THEN DX = -DX : REM WALL
COMING UP?
250 IF POINT(X,Y+DY) =-1 THEN DY = -DY
260 X = X + DX : Y = Y + DY : REM MOVE PIXEL
270 IF X = 11 AND DX = 1 THEN SC = SC + 1 : PRINT@4,
"SCORE =";SC;
280 IF X < 9 THEN PRINT@534, "GAME OVER!"; :
PRINT @594, "PLAY AGAIN (Y/N)"; : INPUT A$ : IF A$ = "Y"
THEN RUN ELSE CLS : END
290 GOTO 180
```

Admittedly, this is far from the most exciting game program ever written. The action of the ball is dull and predictable, always rebounding at right angles, following essentially the same path every time the game is played. And the game itself is so slow that even the most determined player will be bored by the end of the second game. Nonetheless, the program illustrates many of the principles of graphics animation discussed in this book.

This program uses one new command: POINT. Actually, POINT is not a command but a numeric function, like PEEK. (A numeric function is a BASIC keyword that has a

numeric value and may be used in a numerical expression along with numbers and numeric variables.) When we use the POINT function in a program, it must be followed by two numbers, in parentheses, representing the X,Y coordinates of a pixel position on the TRS-80 screen. If the pixel in that position has been turned on, POINT(X,Y) will be equal to −1. If the pixel in that position has not been turned on, or has been turned off, POINT(X,Y) will be equal to 0. Thus, by testing the value of POINT(X,Y), we can determine whether a pixel has been illuminated in a given position on the display.

Let's analyze this program in detail, line by line:

LINE 10: The CLS statement clears the screen, as before.
LINE 20: The player's score will be held in numeric variable SC. The value of this variable is initialized to 0 in this line. (Yes, TRS-80 BASIC will automatically set all numeric variables equal to 0 when we RUN a program, but it is good practice to initialize such variables explicitly at the beginning of the program.)
LINES 30–50: These operations draw the borders at the top and bottom of the screen.
LINES 60–80: Draw the border at the right-hand side of the display.
LINES 90–110: Draw the paddle, which extends initially from coordinates 9,22 to coordinates 9,26.
LINE 120: Prints an initial score of 0 at the top of the display. (See the section starting on page 41 for an explanation of the PRINT @ command.)
LINE 130: Initializes variable PY to a value of 22. PY holds the Y coordinate of the top of our paddle. This coordinate will be changed as we move the paddle up and down with the arrow keys.
LINE 140: Initializes variables X and Y, which contain the X,Y coordinates of the "ball" (that is, the pixel that we will be hitting across the playfield).
LINE 150: Initializes variables LX and LY, which contain the previous ("last") X,Y coordinates of the ball and which will be used to reset the pixel image between "frames."
LINES 160–170: Initialize variables DX and DY, which contain the "directions" that the X and Y coordinates of the ball will be incremented by the animation loop.
LINE 180: PEEKs the keyboard matrix to see if the arrow keys are being pressed.

LINE 190: Checks to see if the down arrow key is being pressed (A = 16). If so, it also checks to see that the paddle hasn't gone all the way to the bottom of the playfield (PY < 42). If both of these conditions are met, it erases the top pixel of the paddle image (at coordinates 9,PY), draws a new bottom pixel (at coordinates 9,PY + 5), and adds 1 to the Y coordinate of the paddle image (PY = PY + 1). The net effect of all this is to move the paddle image down one pixel when the down arrow key is pressed.
LINE 200: Checks to see if the up arrow key is being pressed (A = 8). If so, and if the paddle hasn't gone all the way to the top of the playfield (PY < 4), the paddle image is moved up one pixel, in the same fashion that the previous line moved it down one pixel.
LINE 210: Erases the previous image of the ball, at coordinates LX,LY. The first time through the loop, this will have no effect, because the ball will not have been drawn yet.
LINE 220: Draws the ball, by setting a single pixel at coordinates X,Y.
LINE 230: Sets the values of LX and LY to the current X and Y coordinates, before they are altered in line 260.
LINES 240–250: These lines check ahead to the next X and Y coordinates at which the ball will be drawn, to see if the pixels in these positions have been turned on. (This check is performed by the POINT function, as described above.) If so, it is assumed that the ball is about to collide with a wall, or with the paddle. In order to make the ball seem to bounce, one of the two "direction" variables (DX and DY) is *negated*—that is, given a value that is precisely negative to its current value—like this: DX = −DX or DY = −DY. This reverses the direction in which that coordinate is being incremented, and makes the ball seem to move off in another direction, that is, to "bounce."
LINE 260: The ball is moved, by adding the value of DX to the X coordinate and DY to the Y coordinate. If the value of DX is 1, then the ball will move to the right; if DX is −1, it will move to the left. If the value of DY is 1, then the ball will move down; if DY is −1, it will move up. Since the ball will always be moving in *two* directions at one time—left and up, left and down, right and up, or right and down—it will always move on a diagonal.
LINE 270: Checks to see if the ball is at X coordinate 11 (X = 11) and moving to the right (DX = 1). This is a condition that will only be true if the ball has just bounced off the

paddle; if it is detected, 1 is added to the score (SC = SC + 1) and the revised score is printed at the top of the screen.
LINE 280: Checks to see if the ball is at X coordinate 9. If so, the ball has gone past the paddle and the game is over. A message to that effect is printed and the user is asked if he or she wishes to play another round.
LINE 290: Loops back and does it all again.

SPEEDING UP THE ACTION

Why is this game so slow? Mostly because it's written in BASIC. Considering everything that actually goes on while this program is being executed, it's remarkable that it runs as fast as it does.

BASIC is an *interpreted* language. This means that a BASIC program must first be translated into a form that the computer can understand. This translation is performed by a program called the BASIC interpreter. As each BASIC statement is encountered, the interpreter painstakingly deciphers the specific instructions in that statement and tells the computer how to execute them. This is a time-consuming process. The interpreter spends the great majority of its time just figuring out what you want it to do, and only a very small amount of time actually doing it.

The SET and RESET commands are especially time-consuming. Not only must the interpreter figure out that you are asking it to turn on (or off) a specific pixel on the screen, but it must figure out where that pixel is on the basis of the coordinates that you have given it. This requires a great many calculations for each pixel. Programs that use SET and RESET tend to be slow, much too slow for arcade-style action.

Of course, we can use some tricks to speed up this program. One of them is to add a DEFINT statement. The DEFINT statement tells the computer to treat certain numeric variables within the program as *integer variables.* An integer variable is a variable that can only be set equal to an integer numeric value—that is, a whole number in the range −32768 to 32767. Ordinarily, numeric variables in a Level II BASIC program are assumed to be *single precision variables*—that is, variables that can be set equal to a much

wider range of values, with fractional values. Note that this is a TRS-80 BASIC command and will not necessarily be available in other versions of the language.

Add this line to the program:

```
15 DEFINT A-Z
```

This tells the BASIC interpreter to treat every numeric variable beginning with the letters A through Z (which is to say, every numeric variable) as an integer variable. Because the BASIC interpreter uses much faster arithmetic routines when it deals with integer variables, this will cause a noticeable speedup in the execution of the program. The ball won't suddenly shoot across the screen like a bullet, but it will move faster. We'll use this trick in many of the animation programs discussed later in this book. Note, however, that you had best not use this statement in a program that requires very large, very small, or fractional values. (Fortunately, this program does not, although some other programs in this book, such as the line-drawing routine described earlier in this chapter, do.)

We could also speed up this program with the aid of a *compiler.* A compiler is a program that translates a program written in a high-level language (like BASIC) into a special language called machine language, which the computer finds much easier to execute. Because a machine language program does not have to be translated by an interpreter, the result will be a program that executes many, many times faster than a BASIC program. Several compilers available commercially will translate a Level II BASIC program into machine language. If you do a lot of BASIC programming on a TRS-80, you may want to buy one. After compilation, this program will run so quickly that you may have to insert extra instructions just to slow it down!

In this book, however, we are primarily interested in programs that run in ordinary, interpreted BASIC. And the best way to speed up an animation program written in BASIC is, alas, to avoid slow commands like SET and RESET. There are much faster ways to create animation in Level II BASIC. Although we will return to the SET and RESET commands in later chapters, we are going to turn now to a much faster method of animating graphics on the TRS-80: using the PRINT command.

Suggested Projects

1. The BASIC function ASC is used to convert a character into the ASCII code for that character. For instance, ASC("A") is equal to 65—the ASCII code for the letter "A". ASC(D$) is equal to the ASCII code of the character represented by string variable D$. (If the string represented by D$ contains more than one character, the ASC function will be equal to the ASCII code of the first character in the string.) Write a short program that uses the ASC function in conjunction with the INKEY$ function to determine the ASCII codes produced by every key on the TRS-80 keyboard. Create a chart based on this information.

2. Notice that the SHIFT key does not produce a code number when pressed. Notice also that most keys will produce a different code number when pressed simultaneously with the SHIFT key than when pressed without the SHIFT key. Add these SHIFTed values to your chart.

3. Notice that if you press the left SHIFT key and down arrow key together, then press a third key at the same time, the third key will produce yet anoth-

er code number, completely different from the code it produces normally or when pressed with SHIFT alone. The SHIFT-down arrow combination on the TRS-80 is referred to as the CONTROL key (and has the same effect as the key marked CTRL on certain other computers), because it causes keys on the keyboard to produce the codes for control characters. These codes are always in the range 0 to 31 and are often identical to those produced by control keys such as ENTER and the arrow keys. For instance, CTRL-M (that is, the SHIFT-down arrow combination) pressed simultaneously with the M key produces the code 13. Add these CTRL key values to your chart.

4. Write a program, using the PEEK function, that will explore the keyboard matrix of the TRS-80. Create a chart showing which numbers appear at which locations in the TRS-80's memory when different keys are pressed. (The keyboard matrix lies between memory addresses 14331 and 14464.)

5. Rewrite the arcade game program in this chapter so that it can be played simultaneously by two people. One player could control a paddle at one side of the screen, and the other could control a paddle at the other side of the screen. The object would be to bat the ball back and forth, as in Ping-Pong. When one player misses the ball, the other player scores. A new ball should appear every time a ball is missed, until one player reaches a score of 21.

6. Write a two-player game where both players draw lines on the screen simultaneously, using four keys on the keyboard. (These keys need not be the arrow keys.) The lines should grow continuously in length, with the player controlling the direction of growth from the keyboard. The first player who collides with the border of the screen, the other player's line, or the player's own line loses.

PRINTING A PICTURE

When you learned to program in BASIC, the first command you probably learned was PRINT. And, indeed, PRINT is the most versatile, most obviously useful of BASIC commands. The PRINT command is the method by which the BASIC programmer gets information out of the computer. It is the way a BASIC program speaks to the world. It can be used to output words, sentences, numbers—even the results of equations. But did you know that the PRINT command could be used to create graphics?

Type this command in the immediate mode:

```
CLEAR 1000
```

and press ENTER. Then type this:

```
PRINT@448, STRING$(64,134)
```

and press ENTER again. Instead of printing words or numbers on the display of the computer, this command draws a horizontal line from the left side of the screen to the right. The horizontal line is identical to the one we drew with the SET command in the last chapter, except for two details: it is drawn almost instantaneously, and only one command is required to draw it.

The advantages of the PRINT command over the SET command should be immediately obvious. PRINT is both faster and more efficient than SET. Of course, the SET command can do things the PRINT command cannot, but the reverse is also true. On the whole, the PRINT command is the superior method of creating graphics on the TRS-80, especially in situations where speed and efficiency are paramount.

THE CHARACTER SET

In the last chapter, we spoke about the ASCII code, the system your computer uses for storing characters internally as numbers. Here is a program that will display all of the printable characters (as opposed to control characters) in the ASCII character set, along with their code numbers:

```
10 CLS
20 FOR I = 32 TO 127
30 PRINT I; "("; CHR$(I); ")";
40 NEXT I
```

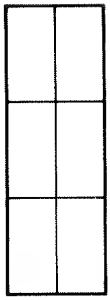
The numbers 32 through 127 will be displayed on the screen, each followed (in parentheses) by the character represented by that number in the ASCII code. We have printed these characters with the aid of the CHR$ function. We have not included the code numbers 0 through 31 because they represent control characters.

Although the ASCII code uses only the numbers 0 through 127, the designers of the TRS-80 elected to use the numbers 128 through 255 to represent additional characters not taken into account by the ASCII code. The numbers 128 through 191 were specifically put aside for *graphics characters.*

Change line 20 of the above program to read:

```
20 FOR I = 128 TO 191
```

and RUN it. This time, next to each number, you will see a small block made up of six pixels, some turned on and some turned off. These are the TRS-80 graphics characters. Imagine them, if you will, as the pieces of a computerized



Pixel arrangement for graphics characters

jigsaw puzzle. We will be using these characters to build pictures, just as the pieces of a jigsaw puzzle are used to build pictures. Every picture that we display using the PRINT command will be made up of some combination of these characters.

By using the CHR$ function, we can induce the PRINT command to display any of these graphic characters, or combinations thereof, on the screen. By combining these characters in imaginative ways, we can create graphic images that look like identifiable figures: spaceships, aliens, flowers, animals, even human beings. And, by adding a few additional instructions to the PRINT command that displays these images, we can even animate them.

To see one simple image that can be created out of these graphic characters, type the following in the immediate mode:

```
PRINT CHR$(154); CHR$(144)
```

A tiny image, constructed of five pixels, will appear at the left-hand edge of the screen, directly below the command itself. Although the image is small and lacking in detail (we will be creating more ambitious images later), you can easily imagine that it represents a miniature spaceship that might be part of a computer arcade game. It is made up of only two of the TRS-80 graphics characters (154 and 144), displayed one after another on the screen, just as two letters of the alphabet might be displayed by the PRINT command.

Using techniques similar to those discussed in the last chapter, we can animate this image. All we need is some

method of placing the image at a specified location on the TRS-80 display and a method of changing this location and redrawing the image, just as we redrew the animated pixel in the last chapter.

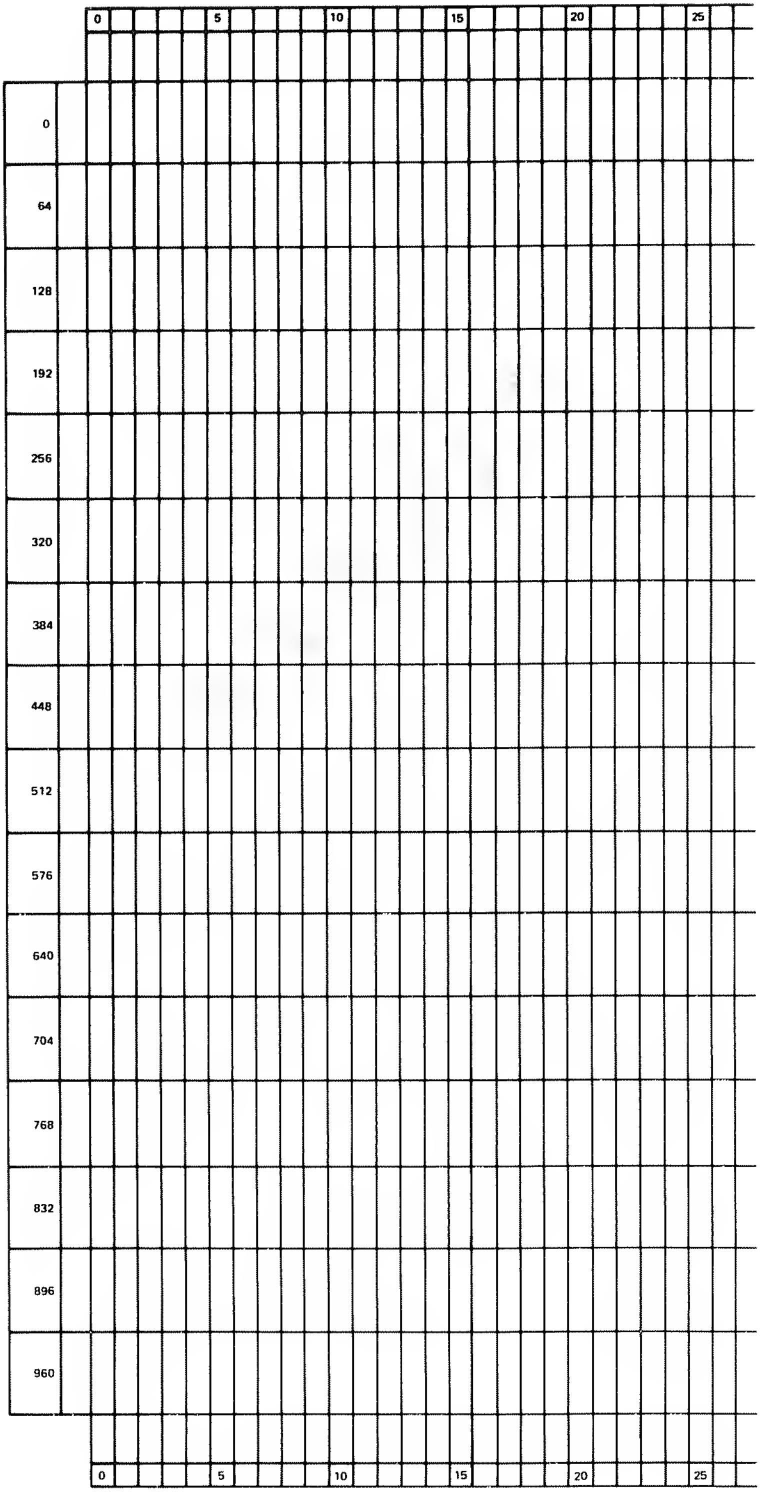
PRINTING IN THE RIGHT PLACE

Precise location of the image is accomplished with the PRINT@ (pronounced "PRINT AT") command. The @ symbol in this command must be followed by a number in the range 0 to 1023, representing the position on the screen where you wish the next character or line of text to be printed. The first position on the screen, in the upper left-hand corner, is given a PRINT@ number of 0; the second position, to the immediate right of this one, is given a number of 1; the last position on the screen, in the lower right-hand corner, is given a number of 1023. The chart on pages 42 and 43 shows all of the PRINT@ numbers for the TRS-80 display.

For a quick demonstration of how PRINT@ works, type the following program:

```
10 CLS
20 INPUT "PRINT WHERE";A
30 PRINT@A, "PRINT HERE!"
40 GOTO 20
```

and RUN it. The screen will clear and you will be prompted to input a PRINT@ position. Type a number between 0 and 1023 and press ENTER. The program will print the words "PRINT HERE!" at the position represented by the number you typed. You will then be prompted, on the following screen line, for another number. Experiment with a wide range of numbers. Large numbers, those close to 1023, will cause printing to occur at the very bottom of the screen. Small numbers, those close to 0, will cause printing to occur near the top. Note that printing on the very bottom line of the screen (positions 960 to 1023) will cause the screen to scroll up one line. Note also that numbers outside of the range 0 to 1023—negative numbers, for instance, or numbers larger than 1023—will cause the program to terminate with an "ILLEGAL FUNCTION CALL ERROR." It's up



PRINT@ *chart. Radio Shack sells "video display worksheets" similar to this.*

		30					35					40					45					50					55					60			63	
																																				63
																																				127
																																				191
																																				255
																																				319
																																				383
																																				447
																																				511
																																				575
																																				639
																																				703
																																				767
																																				831
																																				895
																																				959
																																				1023
		30					35					40					45					50					55					60			63	

to you as the programmer to make sure that you don't use numbers outside of this range with the PRINT@ command. We can take such precautions in the above program by adding this line:

```
25 IF A < 0 OR A > 1023 THEN CLS : PRINT "NUMBER OUT
OF RANGE": GOTO 20
```

Now there is no chance of the program being interrupted by an unsightly and irritating error message.

We can combine the PRINT@ command with our spaceship graphic to produce a program that prints the spaceship smack in the middle of the display, like this:

```
10 CLS : A = 540
20 PRINT@A, CHR$(154) ; CHR$(144);
```

Note that we have placed a semicolon at the very end of this PRINT statement. You may recall from your previous programming experience that the semicolon tells the BASIC interpreter *not* to print a carriage return when the PRINT command is finished. When printing graphics characters on the screen of the TRS-80, it is usually important to include this semicolon, since the TRS-80 carriage return has the added effect of erasing all characters from the current print position to the right-hand edge of the screen, which could have the unfortunate side effect of destroying carefully constructed graphics displays. Although the semicolon is not strictly necessary in this program, it is a good idea to get into the habit of using it, since the absence of the semicolon can often produce subtle but destructive bugs in graphics programs.

RUN the above program. As promised, the spaceship image will appear at PRINT@ position 540, which is directly in the center of the video display.

ANIMATION WITH PRINT@

You may not be terribly impressed by this program, but we're just beginning. Now we can animate the spaceship image, in the same way that we animated the pixel in the previous chapter. Add the following lines to the above program:

```
30 PA = A
40 K$ = INKEY$ : IF K$ = "" THEN 40
50 IF K$ = CHR$(8) THEN A = A -1
60 IF K$ = CHR$(9) THEN A = A + 1
70 PRINT@PA," ";
80 GOTO 20
```

You should recognize the familiar INKEY$ routine that we developed in the last chapter. K$ is set equal to the character produced by the last key pressed on the keyboard. If this is CHR$(8) (the left arrow key), the value of A is reduced by 1, effectively moving the PRINT@ position one space to the left. If the last key pressed produced CHR$(9) (the right arrow key), the value of A is increased by 1, moving the PRINT@ position one space to the right. Variable PA contains the previous value of A, so that we can erase the last image of the spaceship at that position by printing two blank spaces (" ") after it. Line 20 redraws the ship at the new position.

RUN the program. Tap the left and right arrow keys to move the spaceship across the screen. Take special note of what happens when the spaceship reaches the far edge of the screen: it "wraps around" to the opposite edge. For instance, if you move the spaceship all the way to the right-hand side of the screen, it will vanish and reappear at the left-hand side—unlike the animated pixel in the last chapter, which refused to go farther than the edge of the screen (on penalty of an error message). This wrap-around effect is a result of the way the PRINT@ positions are numbered. Unlike the SET coordinates, the PRINT@ positions only reach their maximum and minimum values (0 and 1023) at the upper left-hand and bottom right-hand corners of the screen. Thus, you will only receive an error message if you try to move the spaceship beyond the first or last line of the video display.

Moving the spaceship all the way to the top or bottom of the screen would be a fairly difficult task, anyway. Once again, the INKEY$ function forces us to press the appropriate arrow key every time we wish to move the spaceship one position to the right or left; there is no repeat key function. As before, we can remedy this by scanning the keyboard matrix. Change lines 40 through 60 to read as follows:

```
40 K = PEEK(14400) : IF K = 0 THEN 40
50 IF K = 32 THEN A = A -1
60 IF K = 64 THEN A = A + 1
```

MOVING UP AND DOWN

Now the ship will move smoothly and continuously when the left or right arrow key is held down.

We can also add a feature that allows us to move the ship up and down by pressing the up or down arrow keys. As in the pixel-moving program, this is done by changing the position of the spaceship by an appropriate amount when the up or down arrow key is pressed. However, it is not immediately obvious what the "appropriate amount" is.

To move the pixel up or down in the previous chapter, we changed the value of the Y coordinate by 1. To move the spaceship up or down, we must change the value of the PRINT@ position, represented by variable A. However, if we only change this amount by 1, it will move left or right. If we change it by 2 or 3, it will still move left or right, but by a greater amount. How much must we change the position to move it up or down? Look at the PRINT@ chart on page 00. Compare a position on one line with the position directly above or below that position. What is the numerical difference between the two positions?

The answer, of course, is 64, since there are 64 PRINT@ positions on a single line of the TRS-80 screen. Adding or subtracting 64 from variable A will move our spaceship to the equivalent position on the line below or above. Thus, we can add the following lines to our program:

```
64 IF K = 8 THEN A = A -64
68 IF K = 16 THEN A = A + 64
```

and the ship will move up and down in response to pressing the up and down arrow keys.

You will find that it is now very easy to move the ship past the top and bottom lines of the screen—and thus receive an error message for attempting to print the image in an illegal position. Therefore, we must rewrite the program so that such moves will be intercepted, as we did with the pixel-moving program. Here is the rewritten version:

```
10 CLS : A = 540
20 PRINT@A, CHR$(154); CHR$(144);
30 PA = A
40 K = PEEK(14400) : IF K = 0 THEN 40
50 IF K = 32 THEN DA = -1
60 IF K = 64 THEN DA = 1
70 IF K = 8 THEN DA = -64
80 IF K = 16 THEN DA = 64
90 A = A + DA : IF A < 0 OR A > 1021 THEN A = A -DA
100 DA = 0
110 PRINT@PA, " ";
120 GOTO 20
```

Now the image of the ship will simply freeze in place if you try to move it past the top or bottom line. The direction in which we are going to move the PRINT@ position is contained in variable DA. This value is added to variable A in line 90. We don't check for an out-of-range value of A until we have actually altered its value; if the value *is* out of range, we can easily correct that by subtracting the value of DA once again, restoring the original value of A.

LOTS AND LOTS OF CHARACTERS

By increasing the number of graphics characters that we print, we can create larger and more complex images. The following program, for instance, produces a picture of the space shuttle:

```
10 CLS : CLEAR 1000
20 PRINT CHR$(191) + CHR$(189) + CHR$(144)
30 PRINT STRING$(3,191) + CHR$(180)
40 PRINT STRING$(4,191) + CHR$(189) + STRING$(31,176)
+ CHR$(144)
50 PRINT STRING$(9,191) + CHR$(168) + CHR$(138) +
CHR$(149) + CHR$(191) + CHR$(136) + CHR$(140) +
CHR$(132) + CHR$(191) + CHR$(136) + CHR$(140) +
CHR$(174) + CHR$(149) + STRING$(2,140) + CHR$(170) +
STRING$(12,191) + CHR$(179) + CHR$(187) + CHR$(191) +
STRING$(2,188) + CHR$(176)
60 PRINT STRING$(9,191) + CHR$(186) + CHR$(181) +
CHR$(177) + CHR$(191) + CHR$(186) + CHR$(191) +
```

```
CHR$(181) + CHR$(191) + STRING$(2,179) + CHR$(186) +
CHR$(181) + STRING$(2,191) + CHR$(186) + STRING$(17,191)
+ CHR$(159)
70 PRINT CHR$(130) + CHR$(175) + STRING$(18,191) +
CHR$(143) + CHR$(131)
80 PRINT
```

The image is too large to animate, but it could be used as part of a larger graphics display, or even in the title sequence of a program somehow relating to the space shuttle. (Save this program to tape or disk before moving on, since we will be using variations on it later.)

This program introduces a new BASIC function: STRING$. We use STRING$ when we wish to repeat the same character several times in a single string. It works with any ASCII characters, not just graphics characters. STRING$ must be followed by two numbers, in parentheses. The first represents the number of times we wish to repeat the character, and the second is the character itself, or the code number of that character.

For a couple of quick examples, type the following in the immediate mode:

```
CLEAR 1000
PRINT STRING$(64,"A")
PRINT STRING$(40,191)
```

The CLEAR 1000 statement, which you may have noticed in earlier sample programs (including the shuttle program), is a peculiar convention required in TRS-80 BASIC programs that make extensive use of strings. It reserves space in the memory of the computer for the storage of strings. Because the STRING$ function creates a string of characters and temporarily stores it in the computer's memory, we must reserve space for that string. BASIC will automatically reserve space for 50 characters, but we require slightly more than that. Actually, CLEAR 1000 reserves space for one thousand characters, which is far more than we need, but it's usually a good idea to reserve more space than you will actually use, unless memory is at a premium in your computer. (Note that the CLEAR statement also has the effect of "clearing" the value of all program variables, so always use it at the beginning of the program, before any variables have been initialized.)

The statement PRINT STRING$(64,"A") creates a string of sixty-four "A" 's and prints them on the display. Similarly, the statement PRINT STRING$(40,191) constructs a string made up of forty copies of graphics character 191 and prints it on the display. You may recognize this as the same method we used at the beginning of this chapter to draw a line.

In the shuttle program, STRING$ is used to define long strings of repetitious graphics characters, usually character 191. This is the graphics character for which all six pixels are turned on. It is generally used to define large white areas on the screen. In this case, we are using it to shade in the major part of the body of the shuttle.

REFINING THE TECHNIQUE

Upon typing and running the shuttle program, you may notice that it suffers from several deficiencies. One of them is speed, or the lack thereof. Instead of seeing the shuttle appear all at once on the screen, we actually see it being drawn, with the top part appearing first, the middle part second, and the bottom part last. While this may be sufficient for certain kinds of graphics displays, it would hardly be adequate for an animated program, or for a program that needed to display a large number of images in rapid succession. (You may note, however, that these PRINT graphics are, as promised earlier, a great deal faster than the SET graphics of the previous chapter.)

The second problem is in the program itself. It is large, cumbersome, and awkward to type. Printing long sequences of CHR$ and STRING$ characters seems like an awfully inefficient way to produce graphics.

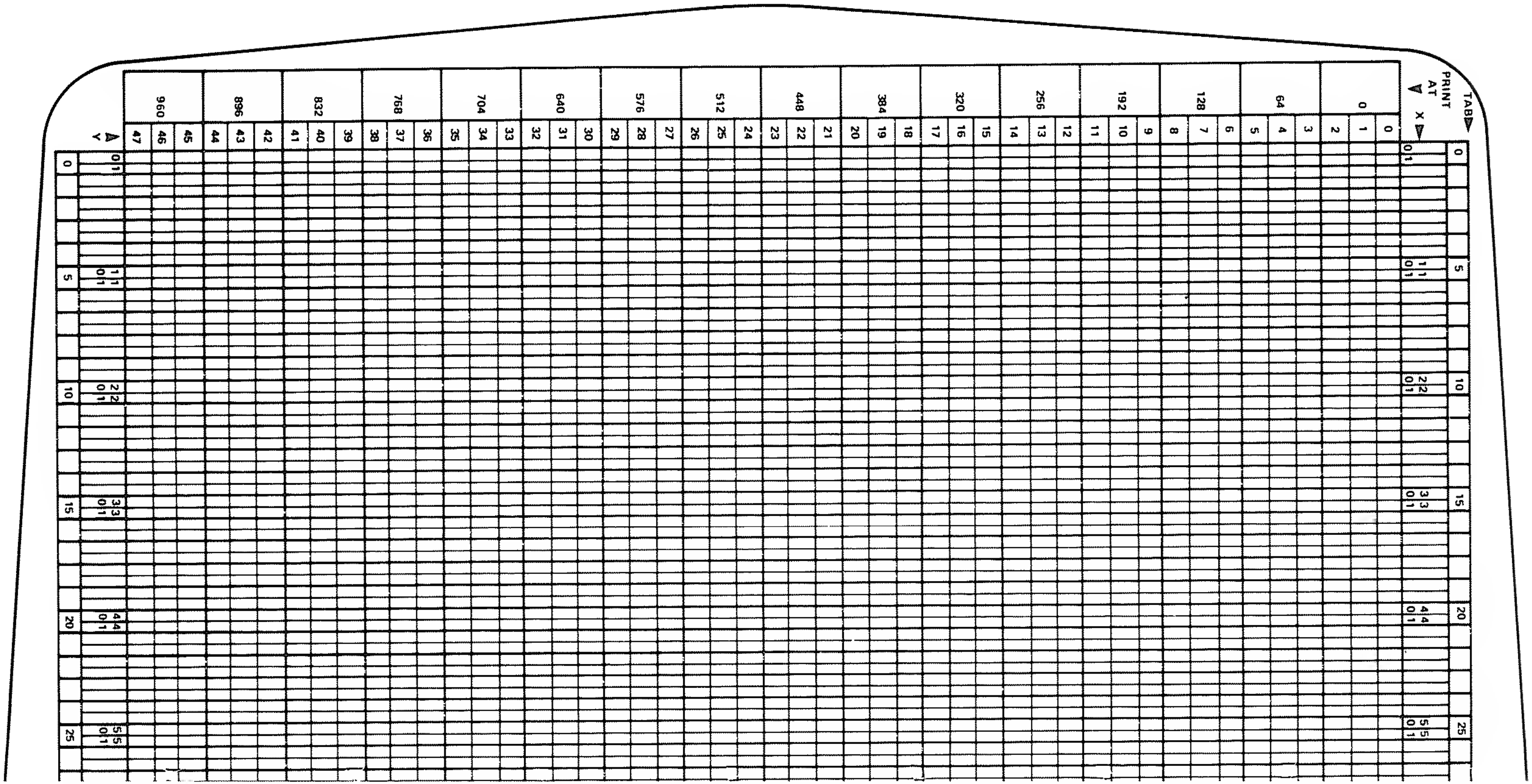
A third problem is that images made up of multiple graphics creations are difficult (and tedious) to design. Is there an easy method by which a programmer can go about creating a picture suitable for a low-resolution TRS-80 graphics display, and then translate that image into a sequence of code numbers?

Probably the most common method of creating TRS-80 graphics is to build up the image block by block on a piece of graph paper. Radio Shack, manufacturer of the TRS-80, sells pads of graph paper specially designed to match the proportions of the TRS-80 display. A reproduction (re-

TRS-80 Video Display Worksheet

Radio Shack

TITLE ________________ PROGRAMMER ________________ PAGE ____ OF ____



30	35	40	45	50	55	60
60 61	70 71	80 81	90 91	100 101	110 111	120 121

Row	
0	
1	63
2	
3	
4	127
5	
6	
7	191
8	
9	
10	255
11	
12	
13	319
14	
15	
16	383
17	
18	
19	447
20	
21	
22	511
23	
24	
25	575
26	
27	
28	639
29	
30	
31	703
32	
33	
34	767
35	
36	
37	831
38	
39	
40	895
41	
42	
43	959
44	
45	
46	1023
47	

60 61	70 71	80 81	90 91	100 101	110 111	120 121
30	35	40	45	50	55	60

duced in size) of one of these sheets is shown on page 50 and 51.

Notice that this graph paper is divided into pixel-shaped rectangles that in turn make up larger rectangles. These larger rectangles, made of six pixels apiece, represent the graphics characters. You can draw a picture on the graph paper by shading in individual pixels with a pencil. You can then use the larger rectangles to determine which graphics characters are necessary to re-create the drawing on the computer.

Around the edges of the graph paper you'll notice a series of numbers. These are the pixel coordinates used with the SET command and the print position numbers used with PRINT@. Once you have designed your picture, you can use these numbers to help determine where on the screen you want to place it.

Graph paper is useful in designing small pictures for use in your programs. However, you may quickly find yourself tiring of this method of graphics creation. Is there a faster, easier way to design graphics?

Yes, there is. In the next chapter, we will see solutions to all of the problems suggested here.

Suggested Projects

1. Write a program that uses the PRINT or PRINT@ command to draw a rectangle in the center of the display.

2. Rewrite the Ping-Pong program from Chapter Two using PRINT graphics rather than SET graphics. See if you can create a substantial increase in the speed with which the program executes.

3. Using graph paper, determine the graphics characters necessary to create a drawing of:

   a. a house
   b. a car
   c. a person
   d. any object ordinarily found around a house

Write a program that will draw these images on the display.

THE GRAPHICS EDITOR

Designing graphics on the TRS-80 or on any other microcomputer can be a tedious occupation. Breaking a picture into a series of character codes, even with the aid of graph paper, can take time and considerable trial and error.

Fortunately, we have at our disposal a device designed for performing tedious and repetitive operations: a computer. We can automate the process of designing graphics.

Of course, graphics design is a creative process, and no creative process can be automated completely—a human being must still be involved. However, the computer can be used for the more mechanical aspects of graphics design, to ease the burden of the programmer.

As promised earlier, here then is a graphics design program, expanded considerably from the sketch program introduced in Chapter One. The internal workings of this program won't be explained—it uses several sophisticated programming techniques that are well beyond the scope of this book, including a machine language routine for saving graphics in memory—but we will discuss in some detail how the program is used. It is not absolutely essential that you use this program in order to appreciate the rest of this book, but it will certainly make it easier for you to experiment with the graphics techniques discussed.

GRAPHICS EDITOR

```
10 CLS
20 DEFINT A−Z : CLEAR 1000 : DIMA%(520), NS, F$, V, G$, J
30 X = 64 : Y = 24 : SF = 0 : PF = 0
40 A$ = STRING$(33,32)
50 V = PEEK(VARPTR(A$) + 1) + 256 * PEEK(VARPTR(A$) + 2)
60 FOR I = V TO V + 33
70 I1 = I : IF I1 > 32767 THEN I1 = I1 −65536
80 READ A : POKE I1, A : NEXT
90 V1 = V + 18 : IF V > 32767 THEN V = V −65536
100 IF V1 > 32767 THEN V1 = V1 −65536
110 DEFUSR1 = V : DEFUSR2 = V1
120 CLS : PRINT TAB(25) "MENU OF OPTIONS"
130 PRINT : PRINT TAB(20) "1. DRAW NEW PICTURE"
140 PRINT TAB(20) "2. RESUME CURRENT PICTURE"
150 PRINT TAB(20) "3. PRINT DATA TO SCREEN"
160 PRINT TAB(20) "4. PRINT DATA TO PRINTER"
170 PRINT TAB(20) "5. SAVE PICTURE TO DISK"
180 PRINT TAB(20) "6. LOAD PICTURE FROM DISK"
190 PRINT TAB(20) "7. RETURN TO BASIC"
200 K$ = INKEY$ : IF K$ < "1" OR K$ > "7" THEN 200
210 ON VAL(K$) GOTO 230, 220, 390, 600, 610, 660, 720
220 V = USR2(0) : GOTO 240
230 CLS
240 SF = 0 : PRINT@980, "PRESS <CLEAR> FOR MENU";
250 DX = 0 : DY = 0
260 P = POINT(X,Y)
270 SET(X,Y)
280 IF (SF = 0 AND P = 0) OR SF = 2 THEN RESET(X,Y)
290 K = PEEK(14400) : IF K = 0 THEN 260 : REM GET LAST
KEY PRESSED. IF NONE, BLINK CURSOR
300 IF K AND 2 THEN V = USR1(0) : GOTO 120
310 IF PEEK(14464) <> 0 THEN SF = 0 ELSE IF K AND 128
THEN SF = 2 ELSE SF = 1 : REM IF SHIFT PRESSED THEN
ENTER CURSOR MOVE MODE ELSE IF SPACEBAR PRESSED
ENTER ERASE MODE ELSE ENTER DRAW MODE
320 IF K AND 8 THEN DY = −1
330 IF K AND 16 THEN DY = 1
340 IF K AND 32 THEN DX = −1
350 IF K AND 64 THEN DX = 1
360 X = X + DX : IF X < 0 OR X > 127 THEN X = X −DX
```

```
370 Y = Y + DY : IF Y < 0 OR Y > 44 THEN Y = Y -DY
380 GOTO 250
390 CLS
400 PRINT "GRAPHICS CHARACTERS:" : IF PF = 1 THEN
LPRINT"GRAPHICS CHARACTERS:"
410 V = VARPTR(A%(0))
420 NS = 0
430 FOR I = 0 TO 14
440 PRINT "LINE #";I+1;":";: IF PF = 1 THEN LPRINT"LINE
#";I+1;":";
450 FOR J = V + I * 64 TO V + I * 64 + 63
460 P = PEEK(J)
470 IF (P = 32) OR (P = 128) THEN NS = NS + 1 : GOTO 500
480 IF NS <> 0 THEN GOSUB 570
490 PRINT " - ";P; : IF PF = 1 THEN LPRINT " - ";P;
500 IF INKEY$ = "" THEN 520
510 IF INKEY$ = "" THEN 510
520 NEXT : GOSUB 570 : PRINT : IF PF = 1 THEN LPRINT
530 NEXT
540 PF = 0 : PRINT "PRESS <CLEAR> TO RETURN TO MENU"
550 IF INKEY$ <> CHR$(31) THEN 550
560 GOTO 120
570 PRINT " -";NS;"SPACE";: IF NS > 1 THEN PRINT "S";
580 IF PF = 1 THEN LPRINT" -";NS;"SPACE"; : IF NS > 1
THEN LPRINT"S";
590 NS = 0 : RETURN
600 POKE 16427, 62 : PF = 1 : GOTO 390
610 V = VARPTR(A%(0))
620 CLS : LINEINPUT "FILENAME? ";F$ : IF F$ = "" THEN 120
630 OPEN "O", 1, F$
640 FOR I = V TO V+959 : PRINT#1, CHR$(PEEK(I)); : NEXT
650 CLOSE#1 : GOTO 120
660 V = VARPTR(A% (0))
670 CLS : LINEINPUT "FILENAME? ";F$ : IF F$="" THEN 120
680 OPEN "I", 1, F$
690 FOR I = 0 TO 3 : LINEINPUT#1, G$
700 FOR J = 1 TO LEN(G$) : POKE V + (J - 1) + I * 255,
ASC(MID$(G$,J,1)) : NEXT : NEXT
710 CLOSE#1 : GOTO 120
720 CLS : END
730 DATA 237, 91, 251, 64, 33, 8, 0, 25, 235, 33, 0, 60, 1, 0, 4,
237, 176, 201
740 DATA 42, 251, 64, 1, 8, 0, 9, 17, 0, 60, 1, 0, 4, 237, 176, 201
```

After you have typed (and carefully proofread) this program, RUN it. The screen will clear and you will see this menu appear on the display:

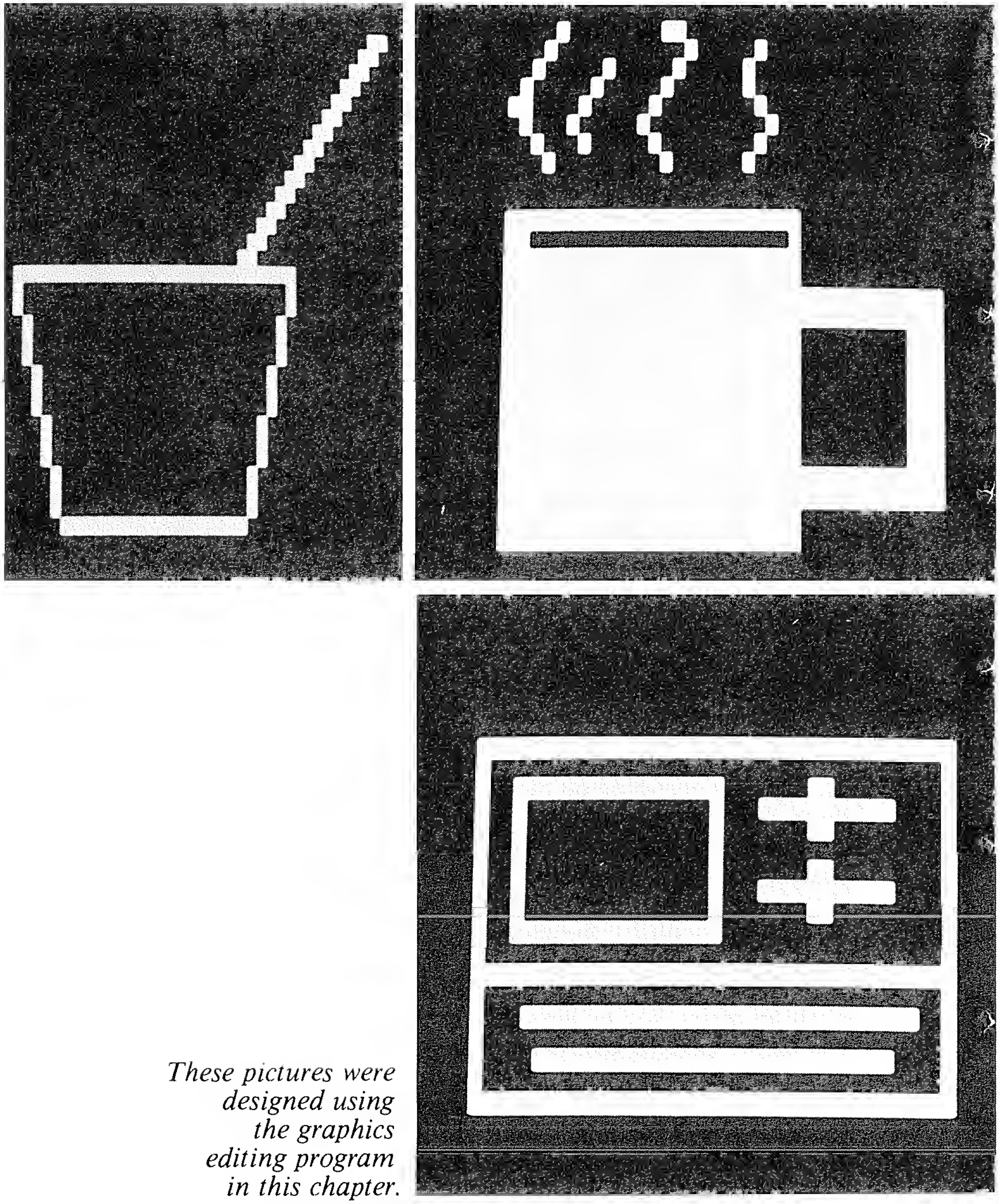
```
MENU OF OPTIONS

1. DRAW NEW PICTURE
2. RESUME CURRENT PICTURE
3. PRINT DATA TO SCREEN
4. PRINT DATA TO PRINTER
5. SAVE PICTURE TO DISK
6. LOAD PICTURE FROM DISK
7. RETURN TO BASIC
```

(If the program doesn't behave as we describe here, go back and check for typing errors once again.) Each of the options in the menu is available at the touch of keys 1 through 7. Touch the 1 key. (There is no need to press ENTER.) The screen will clear again, a flashing cursor will appear in the middle of the display, and the legend "PRESS <CLEAR> TO RETURN TO MENU" will appear at the bottom of the screen.

CREATING PICTURES

You can create pictures with the graphics editor by moving the cursor around the screen. There are three ways to move the cursor, depending on the graphic effect you wish to produce. All involve the use of the arrow keys. If you press an arrow key by itself, the cursor will move in the direction indicated, and will draw a line as it moves, as in the original sketch program. (If you press a vertical key and a horizontal key simultaneously, the cursor will produce a horizontal line.)

If you wish to move the cursor without drawing a line, press an arrow key or keys while simultaneously pressing one of the SHIFT keys. (This maneuver will probably be easiest to perform if you use your thumbs to hold the SHIFT keys and two of your remaining fingers to control the arrow keys.) Finally, if you wish to erase a portion of your drawing, press an arrow key or keys while holding down the space bar (also with your thumbs) and move the cursor over the section you wish to erase.



These pictures were designed using the graphics editing program in this chapter.

In this manner, it is fairly simple to create a drawing on the screen. The size of the drawing doesn't really matter; you can use the graphics editor to create a small drawing, made up of only two or three graphics characters, or a full-screen picture. The bottom line of the screen is unavailable for drawing.

Once you have created a satisfactory image, press the CLEAR key. The screen will be cleared again and the menu will reappear.

Don't worry about your picture. The graphics editor has tucked it away safe and sound in the computer's memory. If you should decide to return to your drawing and do some more work on it, press the 2 key (the "RESUME CURRENT PICTURE" option) and the picture will reappear. On the other hand, if you decide to scrap the drawing and start again, press the 1 key and you will return to a blank graphics screen.

GETTING THE DATA

Assuming that you are satisfied with your drawing as it is, you have several more options. Option number 3 is "PRINT DATA TO SCREEN." Press the 3 key for a demonstration. The computer will display a list of the graphics characters necessary to reproduce the image that you created in the Draw mode. The list is displayed line by line, for all fifteen lines of the picture, with the line number shown at the beginning of each line. A typical data display might look like this:

```
GRAPHICS CHARACTERS:

LINE # 1 : - 64 SPACES
LINE # 2 : - 64 SPACES
LINE # 3 : - 64 SPACES
LINE # 4 : - 10 SPACES - 191 - 189 - 144 - 51 SPACES
LINE # 5 : - 10 SPACES - 191 - 191 - 191 - 180 - 50 SPACES
LINE # 6 : - 10 SPACES - 191 - 191 - 191 - 191 - 189 - 176 -
176 - 176 - 176 - 176 - 176 - 176 - 176 - 176 - 176 - 176 -
176 - 176 - 176 - 176 - 176 - 176 - 176 - 176 - 176 - 176 -
176 - 176 - 176 - 176 - 176 - 176 - 176 - 176 - 176 - 176 -
144 - 17 SPACES
```

LINE # 7 : - 10 SPACES - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 168 - 138 - 149 - 191 - 136 - 140 - 132 - 191 - 136 - 140 - 174 - 149 - 140 - 140 - 170 - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 179 - 187 - 191 - 188 - 188 - 176 - 12 SPACES
LINE # 8 : - 10 SPACES - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 186 - 181 - 177 - 191 - 186 - 191 - 181 - 191 - 179 - 179 - 186 - 181 - 191 - 191 - 186 - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 159 - 12 SPACES
LINE # 9 : - 10 SPACES - 130 - 175 - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 191 - 143 - 131 - 32 SPACES
LINE # 10 : - 64 SPACES
LINE # 11 : - 64 SPACES
LINE # 12 : - 64 SPACES
LINE # 13 : - 64 SPACES
LINE # 14 : - 64 SPACES
LINE # 15 : - 64 SPACES

PRESS <CLEAR> TO RETURN TO MENU

If you press CLEAR, as requested, at the end of the display, you will be returned to the menu.

What can you do with all this information? Well, that depends on what you intend to do with the picture that you have drawn. If you'll look back at the space shuttle program in the last chapter, you'll see that the above printout represents the graphics characters in the space shuttle picture. That picture was designed with the aid of this graphics editor. By studying the earlier program, you can see how the characters printed out above were transcribed into the PRINT statements of the space shuttle program.

Notice that the graphics utility prints out spaces in a special manner. Instead of listing the graphics characters for a blank space (either an ASCII 32 or 128), it simply prints the word SPACES, preceded by the number of spaces. This makes it easier to pass over large blank areas when transcribing this information into a program. Only if the spaces appear directly in the middle of our drawing is it necessary for us to consider them. Thus we may dismiss the first three lines and last six lines of this picture, which contain only spaces, and concentrate on the six lines between

them. These are the six lines that make up the shuttle picture. Notice also that each of these lines begins with 10 SPACES. We may pass over these spaces as well; they are not part of the drawing proper. Neither are the spaces at the end of each line.

The remaining characters must be transcribed into the program in order to reproduce the picture. If you don't wish to copy all of these numbers onto a sheet of paper, and you have a printer attached to your computer, you may choose option number 4 from the main menu: "PRINT DATA TO PRINTER." This will print out exactly the same information on your printer as option number 3 prints on the display.

SAVING THE PICTURE

If at any time you should wish to return to your drawing and make additions or changes, you can still resume drawing with option number 2, even after you have printed out the graphics data with options 3 and 4. Should you wish to preserve your drawing for further work or viewing at a later time, option 5 ("SAVE PICTURE TO DISK") allows you to save the picture to your disk drive. When this option is selected, you will be prompted to supply a filename. This is the name under which the picture will be stored on the disk. Filenames on the TRS-80 can be any length up to eight characters and must contain only letters of the alphabet and numerals (though the first character must be alphabetic). You can also add an extension to the filename, if you wish, by typing a slash (/) at the end of the name and adding up to three more characters. Extensions are typically used for identifying the type of information in a file; for a saved graphic image, you might use an extension such as /GRA or /IMG or /PIC. Once you have input the filename, the file will be saved to the disk in drive #0 and you will be returned to the menu. If you wish to save to a disk in a different drive, add a colon followed by the desired drive number at the end of the filename.

For example, if you wished to call your picture XMASTREE/PIC and save it to drive #1, you would type the following when prompted for the filename:

```
XMASTREE/PIC:1
```

Once you have saved a picture to the disk, it can be reloaded into memory with option number 6, "LOAD PICTURE FROM DISK." Be aware, however, that this will obliterate any picture currently in the program's memory.

Finally, option number 7 ("RETURN TO BASIC") ends the program and puts the user back in the BASIC interpreter. If you should accidentally choose this option before saving your picture (or if you should hit the BREAK key at an inopportune moment, producing roughly the same effect), you can reenter the graphics utility with your picture intact by typing GOTO 120.

A graphics utility such as this provides the programmer with two great advantages: you can see what a picture looks like on the screen of your computer (as opposed to a sheet of graph paper), and you can find out immediately what graphics characters are necessary to re-create the picture in a program, instead of tediously calculating the list of graphics characters from a list of code numbers.

Now that you have such a utility at your disposal, try it out. Sit down at your computer, with the graphics editor running, and design a few pictures. Save them on your disk, or print out a list of the characters on your printer. Incorporate the graphics characters for a small picture into a series of PRINT statements, as we have already done with the shuttle picture, and incorporate those PRINT statements into a program. In time, you will develop a feel for what can be done within the limited resolution of TRS-80 graphics, and for what cannot be done.

Now that you have built up a library of pictures, we will look at ways of using them in your programs.

Suggested Projects

Use the graphics editor to create pictures of:

a. a house
b. a car
c. a person
d. any object commonly found around a house

Compare these to the pictures you created with graph paper in the previous chapter. Are the pictures better? Worse? Was it easier to create them on graph paper or with the graphics editor?

STRINGS AND DATA STATEMENTS

In programming, a *string* is a series of characters—letters of the alphabet, numerals, punctuation marks, and so on—treated by the computer as a single unit. You can print a string to the graphics display, extract portions of it with BASIC commands such as RIGHT$ and MID$, and so forth. If you have been programming long in BASIC, you probably have a good idea of what can be done with strings of characters. Strings are very important in programming, and TRS-80 BASIC offers excellent string-handling facilities.

Generally, we think of strings as representing words and sentences. The following are examples of such strings:

```
"HELLO"
"THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG."
"TOMORROW AND TOMORROW AND TOMORROW, CREEPS
IN THIS PETTY PACE FROM DAY TO DAY."
```

We can use string variables to store and manipulate these variables. For instance, the statement

```
A$ = "AARDVARK"
```

stores the string "AARDVARK" in the computer's memory at the location represented by the string variable A$.

However, we need not restrict the contents of our strings to letters of the alphabet, numerals, and punctuation marks. TRS-80 BASIC allows us to include any kind of character in a string, as long as the TRS-80 provides a code number for that character.

Thus, it is possible to create a string made up of graphics characters. In fact, it is not only possible but extremely useful. Once we have created such a string, we can do the same things with it that we would do with an ordinary string. We can print it on the video display, slice out portions of it with MID$, save it on the disk, and so on.

As an example, type the following statement in the immediate mode:

```
C$ = CHR$(183) + CHR$(179)
```

This sets string variable C$ equal to a two-character string made up of the two graphics characters 183 and 179. (The plus sign [+] links two strings or characters together into a longer string.) Now type the following, in the immediate mode:

```
PRINT C$
```

This prints the string itself on the video display. You should see a small graphic-image, resembling a somewhat enlarged letter "C". Not a particularly spectacular graphic, but perhaps it could be used to represent the main character in a maze game. This "picture" is made up of the two graphics characters concatenated together by the assignment statement above. (When using the plus sign to link strings together, bear in mind that TRS-80 BASIC will not allow you to create a string longer than 255 characters.)

Now press the CLEAR key to clean off the screen, and type the following in the immediate mode:

```
PRINT@540,C$
```

This PRINT@ command places an enlarged "C" right in the middle of the screen. It doesn't take much thought to see that we can use this technique to create an animated sequence.

Type the following program and RUN it:

```
10 CLS
20 C$ = CHR$(183) + CHR$(179)
30 FOR I = 512 TO 575
40 PRINT@I,C$
50 PRINT@I," "
60 NEXT
70 GOTO 30
```

Now the enlarged "C" races from one side of the screen to the other, repeatedly. The image is flickery, but we've seen how to remedy such problems in earlier chapters. In fact, this program produces no effect that we haven't been able to produce through other methods in earlier chapters. However, we are now storing the graphic image in a string variable, and this will greatly enlarge the range of effects that we can achieve, particularly in the area of animation.

SHUTTLE IN A STRING

For instance, you will recall one of the problems we experienced with the space shuttle program in Chapter Three. The image was printed rather slowly, a line at a time. There is a reason for this slowness: every time BASIC encounters the CHR$ function, it must perform a short calculation to decide precisely which character you are asking it to print. Before each line of the shuttle picture was printed, BASIC performed this calculation on every CHR$ in the entire line, assembling all of the necessary graphics characters within the TRS-80's memory, and only then printed the line on the screen. This slowed down the printing of the picture.

However, if we store each line of the drawing in a string variable, these calculations are performed during the assignment statement, rather than the print statement; the print statement therefore executes a great deal faster.

We can rewrite the space shuttle program with the aid of string variables. Modify your copy of this program as follows:

```
10 CLS : CLEAR 1000
20 A$ = CHR$(191) + CHR$(189) + CHR$(144)
30 B$ = STRING$(3,191) + CHR$(180)
```

```
40 C$ = STRING$(4,191) + CHR$(189) + STRING$(31,176) +
CHR$(144)
50 D$ = STRING$(9,191) + CHR$(168) + CHR$(138) +
CHR$(149) + CHR$(191) + CHR$(136) + CHR$(140) +
CHR$(132) + CHR$(191) + CHR$(136) + CHR$(140) +
CHR$(174) + CHR$(149) + STRING$(2,140) + CHR$(170) +
STRING$(12,191) + CHR$(179) + CHR$(187) + CHR$(191) +
STRING$(2,188) + CHR$(176)
60 E$ = STRING$(9,191) + CHR$(186) + CHR$(181) +
CHR$(177) + CHR$(191) + CHR$(186) + CHR$(191) +
CHR$(181) + CHR$(191) + STRING$(2,179) + CHR$(186) +
CHR$(181) + STRING$(2,191) + CHR$(186) + STRING$(17,191)
+ CHR$(159)
70 F$ = CHR$(130) + CHR$(175) + STRING$(18,191) +
CHR$(143) + CHR$(131)
80 PRINT A$ : PRINT B$ : PRINT C$ : PRINT D$ : PRINT E$ :
PRINT F$
```

RUN the program. Notice that the shuttle is now drawn almost twice as fast as before.

However, the six PRINT statements in line 80 take up a lot of space. Is there any way to reduce them to something more compact? The solution is to reduce the six string variables that contain the shuttle image into a single string variable.

It's not hard to envision how to connect all of these strings into a single string variable: we can simply string them together with our old friend, the plus sign (+). However, remember that it's necessary for the cursor to move all the way back to the left-hand side of the screen after drawing each line of the picture. In the current version of the program, this task is performed automatically by the carriage return at the end of each PRINT statement. However, if we were to link the six strings together into a single, long string, we would lose this free carriage return and the shuttle would become a long, meaningless line of graphics characters, stretching all the way across the display. To see this, change line 80 to read:

```
80 S$ = A$ + B$ + C$ + D$ + E$ + F$ : PRINT S$
```

We can avoid this problem by actually embedding the carriage returns within the string. How can we do this? It is

possible, you will recall, to place any characters at all in a string, as long as the computer supplies a code number for that character. This includes control characters, such as carriage returns, which cause cursor movements on the TRS-80 display. The ASCII code for a carriage return is CHR$(13). We can embed carriage returns in our string like this:

```
80 S$ = A$ + CHR$(13) + B$ + CHR$(13) + C$ + CHR$(13)
+ D$ + CHR$(13) + E$ + CHR$(13) + F$ : PRINT S$
```

Now RUN the program. The entire picture of the shuttle will be printed when the PRINT S$ command is executed. All of the graphics characters in the picture, plus the carriage returns necessary to separate the six lines of the image, are stored in string variable S$.

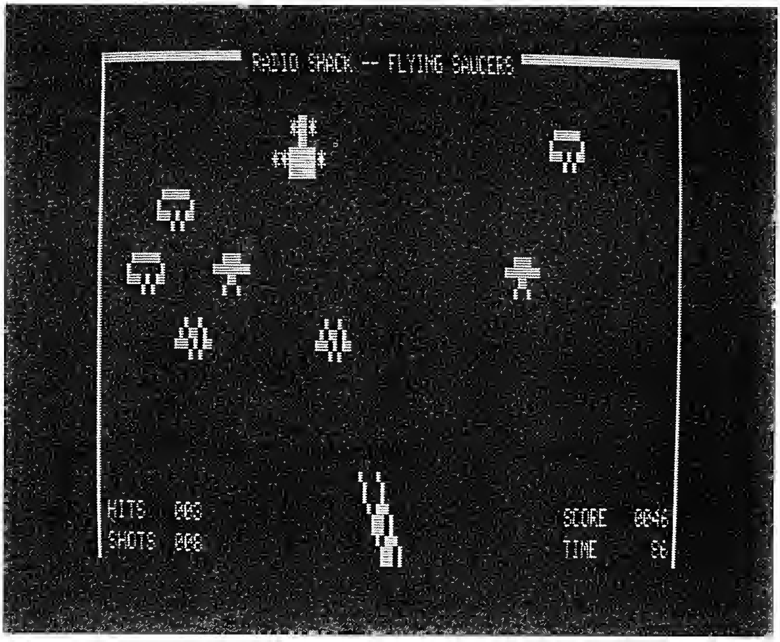
STRINGS AND DATA STATEMENTS

The ability to store an entire picture in a single string variable gives us a flexible tool for manipulating graphic images in our programs, particularly when we need to animate those images. However, the lengthy assignment statements necessary to create those string variables are still awkward. They take up a geat deal of space in the program and require much typing on the part of the programmer (or the poor aspirant who types the programs from the pages of a book such as this). Seven separate assignment statements, two of them covering four lines apiece on the computer screen, are necessary to store the shuttle image in S$. Is there a more efficient way of storing graphics data in a program?

Yes: with the DATA statement. BASIC programmers should immediately recall that the DATA statement is a method of storing string and numeric data within a program, like this:

```
1000 DATA 1, 5, 800, 3129
```

This DATA statement contains four items of data: the numbers 1, 5, 800, and 3129. We can extract this data with the READ statement. When the BASIC interpreter encounters a



TRS-80 video game

READ statement in a program, it searches through all of the program lines until it finds the first DATA statement and stores the first item of data in that statement in the variable following the word READ. For instance, if you wrote a program consisting of these two lines:

```
10 READ A
20 DATA 7, 19
```

the variable A, which follows the word READ, would be set equal to 7, which is the first item in the DATA statement.

If a second READ statement is encountered—or if a second variable is referenced in the first READ statement—that variable will be used to store the second item in the DATA statement. For instance, if we expanded the above program to read:

```
10 READ A, B
20 DATA 7, 19
```

variable A would be set equal to 7 and B would be set equal to 19. We would also get the same effect if we wrote:

```
10 READ A
20 DATA 7, 19
30 READ B
```

The position of the DATA statements in the program relative to the READ statements is irrelevant; only the order is significant. If the computer runs out of data in the first DATA statement and requires more data, it will search for more DATA statements. Thus, we could also write:

```
10 READ A,B
20 DATA 7
30 DATA 19
```

If there aren't enough DATA statements to match all of our READ statements, the interpreter will interrupt the program for an error message.

SHUTTLE IN A DATA STATEMENT

DATA statements are an ideal method of wedging large amounts of graphics information into our programs. Lists of graphics character codes can be contained in DATA statements, and we can put them into string variables with READ statements. Here is our shuttle program one more time, with the shuttle image contained in DATA statements:

```
10 CLS : CLEAR 1000
20 S$ = ""
30 FOR I = 1 to 56 : READ A
40 IF A > 127 THEN S$ = S$ + CHR$(A) : NEXT : GOTO 70
50 IF A = 0 THEN READ B : S$ = S$ + CHR$(B) : NEXT :
GOTO 70
60 READ B : S$ = S$ + STRING$(A,B) : NEXT
70 PRINT S$
80 END
90 DATA 191, 189, 144, 0, 13, 3, 191, 180, 0, 13, 4, 191, 189, 31,
176, 144, 0, 13, 9, 191, 168, 138, 149, 191, 136, 140, 132, 191,
```

```
136, 140, 174, 149, 2, 140, 170, 12, 191, 179, 187, 191, 2, 188,
176
100 DATA 0, 13, 9, 191, 186, 181, 177, 191, 186, 191, 181, 191, 2,
179, 186, 181, 2, 191, 186, 17, 191, 159, 0, 13, 130, 175, 18, 191,
143, 131
```

It may be hard to believe all of the graphics characters in the shuttle picture could be squeezed into two DATA statements, but there they are. Actually, we've used a fairly fancy method of squeezing the characters into these DATA statements, which may look a little confusing at first glance. This isn't the only method of storing graphics characters in DATA statements—you may in fact wish to come up with a method of your own—but it works, so we'll take a detailed look at how this method works.

The READ statement has been placed inside a FOR-NEXT loop, so that it will be executed 56 times, once for each item of data read from the DATA statements. Line 40 checks to see if the last value read (which is contained in variable A) is greater than 127. If so, it assumes this to be the code for a graphics character and adds it to string S$, which was initialized in line 20 as a null string (""). If the character isn't a graphics character, line 50 checks to see if it is a 0. The 0 indicates that the next character is a control character. If the character is a 0, the rest of the IF statement in line 50 reads the next character from the DATA statement, which is the actual control character, and adds that character (rather than the 0) to S$. If the character is neither a 0 nor a number greater than 127, the character is assumed to be the multiplier portion of a STRING$ statement. The next character (the actual graphics character) is READ into variable B, and the two values are used to create the string STRING$(A,B), which is added to S$. This saves us from having to repeat the same character code over and over again in the DATA statement.

When you RUN the program, you'll experience a slight delay while the data is read. This is the price we pay for using DATA statements: a longer period of program initialization. Once the initialization is complete, the picture is drawn at the same speed as before.

The long initialization period is not much of a problem on a program such as this, but on a longer program, with a lot of graphics, reading all of the information from the

DATA statements may take as long as a minute. And, of course, it is in these longer programs with extensive graphics requirements that the DATA statements become crucial.

Ironically, the DATA statement is no longer necessary after the initialization period. It becomes a kind of excess baggage. Is there any way to dispose of the DATA statements after we have run the program for the first time?

STRING PACKING

As a matter of fact, there is. We can use a technique called *string packing*. This is a fairly complex and even slightly dangerous technique—it can destroy your entire program if used incorrectly—but it is the most efficient method possible for storing graphics data within a TRS-80 BASIC program. Whether you choose to use string packing or not is up to you, but it is a powerful programming technique.

String packing makes use of the BASIC POKE command. This command is used to alter the contents of the computer's memory; we can use it to place a string of graphics characters directly into a quoted string within the program. We're not going to go into a lengthy explanation of how this works; rather, you'll be handed an all-purpose string-packing subroutine that will do the job for you. However, you'll be able to see the results of string packing directly in your program, and you may be a little startled when you do.

Here is the shuttle program, back for one last encore, rewritten with a string-packing routine. Before you run this program, proofread it carefully and save a copy on disk or tape! An error in the wrong place could cause disastrous results.

```
5 CLS : CLEAR 1000
7 REM THE STRING IN THE FOLLOWING LINE MUST CONTAIN
EXACTLY 155 CHARACTERS.
10 S$ = "1234567890123456789012345678901234567890l234
5678901234567890123456789012345678901234567890l234567
89012345678901234567890123456789012345678901234567890
12345"
20 V = VARPTR(S$) : M = PEEK (V + 1) + 256 * PEEK(V + 2)
30 FOR I = 1 TO 56 : READ A
```

```
40 IF A > 127 THEN POKE M,A : M = M + 1 : NEXT : GOTO 70
50 IF A = 0 THEN READ B : POKE M,B : M = M + 1 : NEXT :
GOTO 70
60 READ B : FOR J = M TO M + (A -1) : POKE J,B : NEXT :
M = M + A : NEXT
70 PRINT S$
80 END
90 DATA 191, 189, 144, 0, 13, 3, 191, 180, 0, 13, 4, 191, 189, 31,
176, 144, 0, 13, 9, 191, 168, 138, 149, 191, 136, 140, 132, 191,
136, 140, 174, 149, 2, 140, 170, 12, 191, 179, 187, 191, 2, 188,
176
100 DATA 0, 13, 9, 191, 186, 181, 177, 191, 186, 191, 181, 191, 2,
179, 186, 181, 2, 191, 186, 17, 191, 159, 0, 13, 130, 175, 18, 191,
143, 131
```

Once you're certain that everything's okay, RUN the program. There should be a brief pause as the graphics characters are read from the DATA statement, and then the picture of the shuttle will be printed, just as before.

When you LIST the program after running it, however, you might receive a bit of a shock. Moments earlier, line 10 looked like this:

```
10 S$ = "1234567890123456789012345678901234567890123456
78901234567890123456789012345678901234567890123456789O
1234567890123456789012345678901234567890123456789012345"
```

Now, however, it has been mysteriously transformed. The string of 155 numbers that you typed between the quotation marks has been replaced by . . . the picture of the space shuttle!

What exactly has happened here? With the aid of the BASIC POKE command, we have moved the graphics characters from the DATA statements to line 10, where they have been inserted inside the 155-character string that you typed there. Now, in fact, the DATA statements are no longer needed in the program.

To prove this, delete lines 10 through 60 and lines 90 and 100, then RUN the program again. It should work just as before, but with no time-consuming initialization period before the picture is printed. And the program is now only four lines long!

String packing, then, is the most efficient method of storing graphics in a program. Once the string has been packed with the graphics characters, an initialization period is no longer required, and relatively little space is taken up in the program by the graphics.

HOW IT WORKS

Although you needn't understand how string packing works in order to utilize the technique, some readers may be interested in a few of the details. Look back at the original program listing. Notice that in line 20 we use the rather mysterious sounding BASIC function VARPTR. The name of this function is short for "variable pointer"—it points to the place in the computer's memory where the value of a variable is stored. (Actually, in the case of a string variable, it points to another pointer that tells us where the string is stored, but the statement M = PEEK(V+1) + 256 * PEEK(V+2) gives us the address of the actual string.) In the case of our dummy string in line 10, the string is stored within the program itself. The instructions on lines 40 through 60 use the POKE command to place the graphics characters inside this string, so that they become a permanent part of the program. Once stored there, the DATA statements become irrelevant. And because they are now stored in a more efficient form than they were in the DATA statements, the program becomes considerably shorter when the DATA statements are eliminated.

Here is an all-purpose string-packing subroutine that you can use in your programs:

```
1000 REM         ************************************************
1010 REM * STRING PACKING ROUTINE:                              *
1020 REM * V must contain the 'VARPTR'                          *
1030 REM *     address of the string to                         *
1040 REM *     be packed.                                       *
1050 REM * NC must contain the number                           *
1060 REM *     of characters in the string,                     *
1065 REM *     not counting multipliers.
1070 REM * The next DATA statement in                           *
1080 REM *     the program must contain                         *
1090 REM *     the graphics characters to                       *
1100 REM *     be packed.                                       *
1110 REM          ***********************************************
```

```
1120 REM
1130 V = PEEK(V + 1) + 256 * PEEK(V + 2)
1140 FOR I = 1 TO NC : READ A
1150 IF A > 127 THEN POKE M,A : M = M + 1 : NEXT :
RETURN
1160 IF A = 0 THEN READ B : POKE M,B : M = M + 1 : NEXT
: RETURN
1170 READ B : FOR J = M TO M + (A -1) : POKE J,B : NEXT
: M = M + A : NEXT
1180 RETURN
```

You can call this routine from any point within a program with the GOSUB command. Instructions for calling the routine are in the REM statements at the beginning: variable V should equal the VARPTR address of the string to be packed. (This is calculated through the simple statement V = VARPTR(A$), where A$ is the string to be packed.) The string itself should be placed elsewhere in the program, in an assignment statement, followed by a dummy string in quotation marks, like this:

```
A$ = "THIS IS A DUMMY STRING"
```

The dummy string should contain exactly as many characters as there are graphics characters to be packed into it. The number of graphics characters should be in variable NC, where characters to be multiplied (see next paragraph) count as a single character.

The next DATA statement to be READ from should contain the graphics characters, in the same format used in our program that demonstrated the use of DATA statements—that is, all numbers greater than 127 should be graphics characters. Numbers less than 128 (except for 0 and control characters) should be multipliers, that is, they should indicate that the following number is a graphics character to be repeated a certain number of times. For instance, the number sequence

```
100 DATA 5, 143
```

would indicate that the graphics character 143 is to be repeated five consecutive times. As noted above, characters to be multiplied only count as a single character in calculating the value of NC. (However, this character would count

five times when calculating the length of the dummy string.)

Finally, the number 0 indicates that the number following it is a control character. You can use this to insert a carriage return into a graphic, although other control characters can be inserted as well.

If you use the string-packing technique in any of your programs, it is important that you save a copy of the program before you run it. Because string packing tampers with the contents of your computer's memory, you run the risk of accidentally destroying important information contained in the computer's memory, such as the program itself. This will not harm your computer in any way, but you may be forced to press the RESET key to regain control of the computer. In this event, you will want to have a safe copy of the program tucked away, so that you can reload it into the computer and proofread it carefully to see what went wrong.

WARNING: Once you have created a packed string in a TRS-80 BASIC program, you should not attempt to edit the program line containing the packed string, or bizarre results may ensue.

Suggested Projects

Take the programs that you created to draw the house, the car, the person, and the household object, and rewrite them using DATA statements. Then rewrite them using string packing.

ANIMATION ARRAYS

Now that you know how to insert graphics into your programs, and have a tool to facilitate graphics design, let's talk about some of the things that you can do with those graphics. Let's talk about making them move.

Oh, we've talked about animation in earlier chapters, which is to say that we've made a pixel or a picture move from one side of the screen to the other, sometimes in response to a push of one of the arrow keys.

But that's not all there is to animation. When a living creature moves, it doesn't just glide stiffly across your field of vision. Rather, it moves in a symphony of carefully coordinated motions. A human being, for instance, walks across a room with legs and arms swinging, eyes darting, head moving from side to side. A bird soars across the sky with wings flapping, beak opening and closing. Even PacMan (not *quite* a living creature) opens and closes his mouth as he darts about his maze, eating blue dots.

We can add this sort of complex animation to our BASIC programs, at the expense of slightly more complicated graphics techniques. The expense is worth it. Here, for instance, is a short program that shows a PacMan-like creature (actually the enlarged "C" from Chapter Five) chewing its way through a line of dots.

```
10 CLS
20 X = 0
30 C$(0) = CHR$(183) + CHR$(179)
40 C$(1) = CHR$(157) + CHR$(140)
50 FOR I = 512 TO 575 : PRINT@I, ".";  : NEXT
60 FOR I = 512 TO 575
70 X = X + 1 : IF X > 1 THEN X = 0
80 PRINT@I -1," "
90 PRINT@I ,C$(X)
100 FOR J = 1 TO 40 : NEXT
110 NEXT
120 PRINT@I-1, " "
130 GOTO 50
```

And here, in a more elaborate variation on the same theme, a small spaceship glides slowly across the screen:

```
10 CLS
20 X = 0
30 C$(0) = CHR$(174) + CHR$(179) + CHR$(179) + CHR$(132)
40 C$(1) = CHR$(166) + CHR$(183) + CHR$(179) + CHR$(132)
50 C$(2) = CHR$(166) + CHR$(187) + CHR$(179) + CHR$(132)
60 C$(3) = CHR$(166) + CHR$(179) + CHR$(183) + CHR$(132)
70 C$(4) = CHR$(166) + CHR$(179) + CHR$(187) + CHR$(132)
80 FOR I = 512 TO 572
90 X = X + 1 : IF X > 4 THEN X = 0
100 PRINT@I -1, " "
110 PRINT@I, C$(X)
120 FOR J = 1 TO 50 : NEXT
130 NEXT
140 PRINT@I -1, " "
150 CLS : GOTO 80
```

In both cases, there is more to the animation than the simple movement of an image from one side of the display to the other. The PacMan-like creature opens and closes his mouth as he chomps on the dots; a large "eye" on the front of the spaceship rotates ominously from side to side. This animation is more realistic, and more complex, than the animation we created earlier.

The secret is in our use of variable arrays. You should recall, from your earlier programming experiences, that an

array is a method of storing lists of information in a program, in a meaningful order. An array variable is identified by the *subscript* that follows it in parentheses. The presence of the subscript tells us that the variable is only one in an array of variables; the number in the parentheses tells us which element within the array this variable represents. If the subscript is itself a variable or an arithmetic expression rather than a number, then the variable may represent any of the elements in the array, depending on the current value of the variable or expression.

In the two programs above, instead of storing a single image in a single variable, we have stored several different images in an array of variables. The image in each element of the array represents a different stage in the animation. In the first of the two programs, for instance, variable C$(0) contains the image of the creature with his mouth open, while variable C$(1) holds the image of the creature with his mouth closed.

Line 90 of this program prints string variable C$(X) on the display. Because the subscript of this variable is itself a variable, C$(X) may represent either C$(0) or C$(1), depending on the current value of X. Line 70 causes the value of X to alternate between 0 and 1, on consecutive executions of the loop. Thus, as the program executes, we see the image alternate rapidly between the picture of the creature with his mouth open and the picture of the creature with his mouth closed.

Similarly, in the second program, we store the images of the spaceship in variables C$(0), C$(1), C$(2), C$(3), and C$(4). In each, the rotating eye has moved to a new position. Lines 90 and 110 cycle through the five images, displaying each in turn, producing the rotation effect.

And that's the whole trick of complex animation: create as many different images as you need, place them in an array, and move them around the screen with a program loop, shuffling through the elements of the array as required. If you wish to have your animated character vary its movements in different situations, then create several different arrays, or move through the elements of the one array in a different order. For instance, if we wish to reverse the rotation of the spaceship's "eye," we need merely reverse our progress through the array. The only change necessary is in line 90, which would be rewritten like this:

```
90 X = X - 1 : IF X < 0 THEN X = 4
```

It shouldn't be hard to think up nearly endless possibilities for games, written in BASIC, that would utilize these graphic techniques. And the use of such animation need not be restricted to games. The title screen of any program could be enhanced by animated characters, and you may even wish to produce animation for its own sake—comic strips on the computer display, complete with captions and dialogue balloons.

For instance, add these two lines to the spaceship program:

```
75 A$ = "GALACTIC INVADERS"
105 IF I > 536 AND I < 554 PRINT@I -1, MID$(A$,I -536,1);
```

and RUN it. Now, the spaceship drifting from left to right across the screen magically writes the words "GALACTIC INVADERS" in the center of the display. This sequence would make a diverting title screen for an arcade game of that name.

BATTLE IN OUTER SPACE

As an example, here is a listing for a simple arcade game that makes use of almost all of the animation techniques discussed so far:

```
10 REM
20 REM **** COSMIC WAR: A GAME ****
30 REM
40 CLS : DEFINT A-Z
50 C$(0) = CHR$(168) + CHR$(137) + CHR$(134) + CHR$(148)
60 C$(1) = CHR$(160) + CHR$(134) + CHR$(140) + CHR$(137)
+ CHR$(144)
70 S$ = CHR$(168) + CHR$(189)
75 REM *** TITLE SEQUENCE ***
80 N$ = "COSMIC WAR"
90 C = 0
100 FOR I = 512 TO 572 : PRINT@I -1, "      "; : PRINT@I,
C$(C);
110 C = C+1 : IF C > 1 THEN C = 0
120 FOR J = 1 TO 30 : NEXT
```

```
130 IF I > 539 AND I < 556 THEN PRINT@I -1,
MID$(N$,I-539,1);
140 NEXT
150 FOR I = 1 TO 1000 : NEXT : CLS
155 REM *** THE GAME ***
160 P = 990
170 I = 512 + RND(55)
180 C = C + 1 : IF C > 1 THEN C = 0
190 PRINT@I-1, "      "
200 PRINT@I, C$(C)
210 PRINT@P, "   ";
220 P = P + DP : PRINT@P, S$; : DP = 0
230 K = PEEK(14400) : IF K = 0 THEN 270
240 IF K AND 128 THEN X = (P - 959) * 2 : FOR Y = 44 TO 24
STEP -1 : IF POINT(X,Y) = -1 THEN Z = I : GOTO 310 ELSE
SET(X,Y) : RESET(X,Y) : NEXT
250 IF K AND 32 THEN DP = -1 : IF P + DP < 940 THEN DP
= 0
260 IF K AND 64 THEN DP = 1 : IF P + DP > 1021 THEN
DP = 0
270 IF RND(3) = 1 THEN X = (I - 512) * 2 + RND(4) -1 : FOR
Y = 30 TO 47 : IF POINT(X,Y) = -1 THEN Z = P : GOTO 310
ELSE SET(X,Y) : RESET(X,Y) : NEXT
280 IF RND(5) = 1 THEN 180 ELSE I = I +1 : IF I < 572 THEN
180
290 PRINT@I-1, "      "
300 I = 512 : GOTO 180
310 FOR K = 1 TO 30 : PRINT@Z,CHR$(RND(64)+127);
CHR$(RND(64)+127); CHR$(RND(64)+127); CHR$(RND(64)
+127); NEXT
320 CLS : IF Z = I THEN PRINT@537, "CONGRATULATIONS!"
ELSE IF Z = P THEN PRINT@540, "WHOOPS!"
330 PRINT@600, "PLAY AGAIN (Y/N) ?"
340 K$ = INKEY$ : IF K$ = "" THEN 340
350 IF K$ = "Y" THEN 160
360 IF K$ = "N" THEN END
370 GOTO 340
```

The game play is simple: you control a tiny missile base at the bottom of the screen. Pressing the left and right arrow keys moves the base left and right. Pressing the space bar fires a missile (actually, a single pixel) toward the middle of the screen, where an animated alien spacecraft is moving

from left to right. The spacecraft is dropping bombs at random. If one hits you, you will be destroyed; the game will end with the message "WHOOPS!" and you will be given a chance to play again. If you hit the spacecraft, it will be destroyed. The game will end with a message of congratulations and you will also be given a chance to play again.

Before this action begins, you will see a title sequence similar to the one demonstrated.

If you examine the program listing, you will see that this program uses almost every animation technique discussed in this book. You should be able to puzzle out most of the program's operation based on what you've learned so far. Nonetheless, here is a line-by-line explanation of how the program works:

LINE 40: Clears the screen and declares all variables as integers to speed up the action.
LINES 50-70: Defines the two animated images—the spacecraft (in variables C$(0) and C$(1)) and the missile base (in variable S$).
LINE 80: Initializes title to be displayed during title sequence.
LINE 90: Initializes the subscript for the spacecraft array.
LINE 100: Moves the image of the spacecraft from the left side of the display to the right, for title sequence.
LINE 110: Alternates the subscript (variable C) of the spacecraft array between 0 and 1, to display the two images of the spacecraft.
LINE 120: Delay loop. Prevents animation from proceeding too quickly. (Delete this line and see what happens.)
LINE 130: Prints the title characters as the spaceship crosses the center of the screen.
LINE 140: End of loop.
LINE 150: Delay loop to hold title on screen for a few seconds.
LINE 160: Starting position (in variable P) for missile base.
LINE 170: Starting position (in variable I) for spacecraft; 512 is the PRINT@ position at the beginning of the line on which the spacecraft is positioned. The BASIC function RND(55) returns a random number between 1 and 55. This is added to 512 to give the spacecraft a random starting position on that line.

LINE 180: Alternates the two elements of the spacecraft array.
LINE 190: Clears the previous image of the spacecraft.
LINE 200: Prints the image of the spacecraft in its current position.
LINE 210: Clears the previous image of the missile base.
LINE 220: Calculates the next position of the missile base using variable DP ("direction of P"), prints the base at this position and clears DP to 0.
LINE 230: Scans the keyboard matrix to see if a control key is being pressed. If none is found, skips to line 270.
LINE 240: Checks to see if the space bar is being pressed. If so, it fires a missile upward. The expression X = (P −959) * 2 translates the PRINT@ position (P) of the spacecraft into an X coordinate of the type used by the SET command. The FOR-NEXT loop then SETs and RESETs a succession of pixels from the top of the missile base to the middle of the screen (Y coordinates 24–44). The POINT function checks to see if the coordinates where the missile is to be drawn are currently occupied by a pixel. If so—that is, if POINT(X,Y) = −1—it is assumed that the "missile" has collided with the spacecraft and branches to the explosion routine in line 310.
LINES 250–260: Check for left and right arrow keys. If one is pressed, DP (the value added to the position of the missile before it is drawn) is set to 1 or −1 (for right and left, respectively). The value is checked for out of bounds, and DP is cleared to 0 if necessary.
LINE 270: Drops a bomb from the spacecraft. Because the bomb is a random event, it is only executed if the value of RND(3) is 1. This gives the bomb a one-out-of-three chance of being dropped on any pass through the loop. The routine that draws the falling bomb is similar to the routine that draws the missile in line 240: it calculates the X coordinate based on the I position of the spacecraft (but this time it also adds in the value of RND(4)−1, adding an element of surprise to the precise position of the bomb relative to the body of the spacecraft). If the POINT function detects a collision, the program branches to the explosion routine in line 310.
LINE 280: On most executions of the loop, this line will add 1 to the position of the spacecraft, moving it to the right, and will then branch back to line 180 to continue the main

loop of the game if the spacecraft has not moved off the right side of the screen. If RND(5) = 1, however, the position of the spacecraft is not incremented, though the loop is repeated. This slows the speed of the craft ever so slightly, allowing your missile base to catch up with it if you are pursuing it from the left. Otherwise, you would only be able to attack the spacecraft from the right.

LINE 290: Erases the final image of the spacecraft at the right-hand side of the screen.

LINE 300: Moves the spacecraft back to the left side of the screen and continues the main loop.

LINE 310: Draws an explosion on the screen. This is the routine called when a missile strikes the spacecraft or a bomb strikes the missile base. It prints an animated explosion at position Z on the display. If this routine is called when a missile strikes the spacecraft, Z will equal the current position of the spacecraft. If this routine is called when a bomb strikes the missile base, then Z will equal the current position of the missile base. The explosion is generated by drawing a sequence of randomly chosen graphics characters on the screen in a FOR-NEXT loop. (A random-graphics character is chosen with the function CHR$(RND(64)+127).

LINE 320: Clears the screen and prints "CONGRATULATIONS!" if the spacecraft is destroyed and "WHOOPS!" if your missile base is destroyed.

LINE 330: Offers the player a chance to play again.

LINES 340–370: Checks the keyboard for a keypress. If Y is pressed, restarts the game. If N is pressed, ENDs the program. If neither is pressed, checks again.

CUSTOMIZING THE GAME

Cosmic War is not a full-featured game—it is not intended to be—but it could be expanded. Readers are encouraged to do so. For instance, you might wish to add a score routine, such as the one included in the earlier Ping-Pong game. The score could be printed at the top of the screen or at any other location you might choose. Instead of ending the game when a missile hits the spaceship, you could add 1 (or 10 or 100) to the player's score and introduce another spacecraft. (The second spacecraft could be identical to the first, or completely different. Figuring out how a different space-

craft graphic could be introduced into the game without unnecessarily expanding the size of the main program loop is a problem left to the reader. Several solutions are possible.) You could allot the player a certain number of missile bases—three or four, perhaps—and terminate the game only when they are all used up. The number of bases remaining could be displayed alongside the score. Perhaps you could even include an option whereby the user could select the number of bases desired.

Finally, you could tinker with the graphics. If you don't like the way the spacecraft and the missile base are designed, change them. Use the graphics editor program to create new ones. Insert your own graphics characters into the assignment statements in lines 50 through 70. If you feel particularly ambitious, increase the number of images used to depict the spacecraft and the missile base. Increase the number of "frames" in the spacecraft animation from two to three or four or even five. Give the missile base a changing sequence of images.

Nothing in the program as currently written is sacred. Use it as a springboard for any graphics ideas that you might wish to play around with. Customize it. Turn it into your own game.

Or, better yet, start a game of your own design from scratch.

Suggested Projects

1. Write a program that depicts a person walking from one side of the screen to the other.

2. Add an option to the program described in the first project that allows you to guide the person around the display with the arrow keys.

3. Invent your own comic strip, complete with characters, situations, and dialogue balloons or captions.

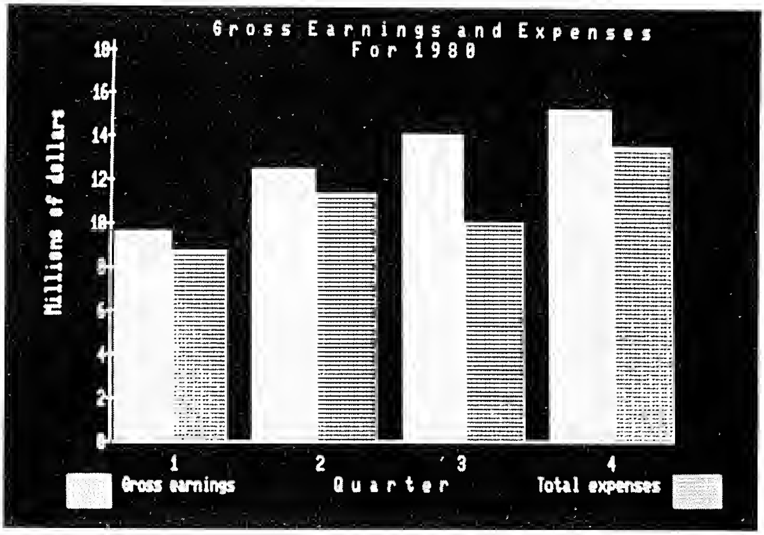
4. Make up your own arcade or video game. Try a simple game, so that the BASIC program will have only a small number of moving objects to control.

THE BUSINESS OF GRAPHICS

There is more to computer graphics, of course, than arcade games and pretty pictures. There are a number of very practical reasons for putting drawings on the screen of your computer.

A picture, they say, is worth a thousand words. On the computer, at least, a picture can be worth a thousand numbers. We can make numbers, and the relationships among numbers, far more meaningful by translating those numbers into a graphic representation.

In Chapter One we saw how the SET command could be used to create bar charts. Such charts can be used to compare numerical values on the display, in a very immediate, visual sort of way. For instance, a company that sells vacuum cleaners might use such a chart to compare its sales figures (expressed in dollars or in numbers of vacuum cleaners) from one year to the next. By displaying ten bars, each representing the number of vacuum cleaners in a given year, it will be immediately apparent which years were best for vacuum cleaner sales, and whether there is an upward or downward trend in these sales. Similarly, if the company sells several different types of vacuum cleaners, each bar could represent the dollar volume of sales for a single type. The relative sales for each type would then be visually obvious.



Bar chart on a TRS-80

THE BAR CHART PROGRAM

Here is a program that will create bar charts to order. You need merely type in how many bars you wish to display (from 2 to 15), what unit of measure you intend to use (dollars, inches, tons, and so on) and how many units each bar should represent. The program will create and display an appropriate chart on the screen:

```
10 CLEAR 1000
20 DIM L(15) : HB = 0
30 CLS : INPUT "TITLE OF CHART";T$
40 INPUT "NUMBER OF BARS DESIRED (2-15)";NB : IF NB <
2 OR NB > 15 THEN 40
50 INPUT "NAME OF STANDARD UNIT (PLURAL; 10 CHARS
OR LESS)";SU$ : IF LEN(SU$) > 58 THEN 50
60 FOR I = 0 TO NB -1 : PRINT "LENGTH OF BAR NUMBER";I
+ 1;"IN STANDARD UNITS"; : INPUT L(I) : IF L(I) < 1 THEN 60
70 IF HB < L(I) THEN HB = L(I)
80 NEXT
```

```
90 CLS : W = 128/NB : SU = 34/HB
100 PRINT@32 - LEN(T$)/2, T$
110 PRINT@96 - (7 + LEN(SU$))/2, "(UNIT = ";SU$;")";
120 FOR I = 0 TO NB -1
130 FOR Y = 44 TO 44 -SU * L(I) STEP -1
140 FOR X = I * W TO I * W + W -2
150 SET(X,Y)
160 NEXT : NEXT : NEXT
170 PW = 64/NB
180 FOR I = 0 TO NB -1
190 I$ = STR$ (I + 1) : I$ = RIGHT$(I$,LEN(I$)-1)
200 PRINT@960 + I * PW + PW/2 -1,I$;
210 NEXT
220 FOR I = 0 TO NB -1
230 L$ = STR$(L(I) : L$ = RIGHT$(L$,LEN(L$)-1)
240 PRINT@832-INT(L(I)*SU/3) * 64 + I * PW + PW/2 -
LEN(L$)/2,L$;
250 NEXT
260 K$ = INKEY$ : IF K$ = ""THEN 260
270 GOTO 30
```

RUN the program. When you are prompted for the name of the chart, type "SALES FIGURES". For number of bars, type "12". When you are prompted for a standard unit, type "MILLIONS OF DOLLARS". Input a number between 0 and 100 for each bar as prompted, then sit back and watch.

The bars will be drawn, one after another, from the left side of the screen to the right, with the individual bars growing upward from the bottom of the screen to the top. After the bars have been drawn, they will be numbered sequentially, on the bottom line of the screen. Then the number of standard units represented by each bar will be printed just above the top of each bar.

When you have tired of looking at the chart created from your information, press any key and the program will prompt you for data to create another chart. Make up any kind of data that you wish: base a chart on your most recent report card (with 5 units representing an "A," 4 units a "B," 1 unit an "F," and so on) or on the amount of money you have earned shoveling snow or mowing lawns in the last three months.

Here is a line-by-line explanation of how the program works:

LINE 10: Opens up space for the strings used by the program.
LINE 20: Dimensions an array called L, which will contain the lengths of the bars in the chart. Initializes the value of HB ("highest bar"), which will contain the length of the largest bar in the chart. (This is used later for establishing the number of units represented by each pixel.)
LINE 30: Clears the screen and prompts for the title to be given the chart. (Title is stored in T$.)
LINE 40: Establishes the number of bars to be included in the chart and checks to make sure that the value is in range. (The number of bars is stored in NB.)
LINE 50: Inputs the name of the unit of measure to be used in the bars. (The name is stored in SU$.) Checks to make sure that the unit name will fit on the display when the chart is drawn. If not, it prompts for another name.
LINE 60: Inputs the lengths of the bars and stores them in L(I). Because the input loop is based on the value of NB, this will prompt only for the number of bars input by line 40.
LINE 70: Establishes the value of HB. If HB is smaller than the value of the current bar, then it is assigned the value of the current bar. This assures that HB is always equal to the length of the largest bar yet assigned a length.
LINE 80: End of loop.
LINE 90: Clears the screen. Establishes the width (W) of each bar in pixels. Since the entire horizontal dimension of the screen (128 pixels) is available for the width of the bars, the width of the individual bars can be calculated by dividing 128 by the total number of bars (NB). Naturally, the more bars on the screen, the fewer pixels in the width of each bar. Also establishes the number of pixels that will represent the standard unit; this value is stored in variable SU. It is assumed in this line that the longest bar on the screen will stretch across the entire vertical area allotted for bars. Thus, the pixel size of the standard unit equals the size of this vertical area (34 pixels) divided by the number of units in the longest bar (HB).
LINE 100: Prints the title (T$) in the center of the top screen line. The statement PRINT @ 32 −LEN(T$)/2, T$ centers the title by subtracting half the length of T$ from 32 (which is the halfway point on the first line) and printing T$ at the resulting position.
LINE 110: Prints the name of the standard unit in the center

of the second line of the display. The method used for centering the text is similar to that in line 100, but slightly more complicated because of the patchwork nature of this string.

LINE 120: Start of the loop that will draw NB bars on the display.

LINE 130: Begins the loop that draws the vertical dimension of each bar.

LINE 140: Begins the loop that draws the horizontal dimension of each bar.

LINE 150: Sets the actual pixels in the bars.

LINE 160: Terminates all three loops.

LINE 170: Calculates the width of each bar in PRINT positions. Stores this value in PW, for reference by later program lines.

LINE 180: Begins the loop that prints a number (from 1 to NB) underneath each bar.

LINE 190: Converts the number for each bar into a string, then removes the trailing blank from the string. BASIC automatically adds a blank space after each number, and a blank space in front of all positive numbers (to occupy the position taken by the minus sign in a negative number). These extra spaces tend to crowd the bottom line of the display, so the leading space is removed by this line.

LINE 200: Centers each number under the appropriate bar.

LINE 210: Terminates the loop.

LINE 220: Begins the loop that prints the unit number at the top of each bar.

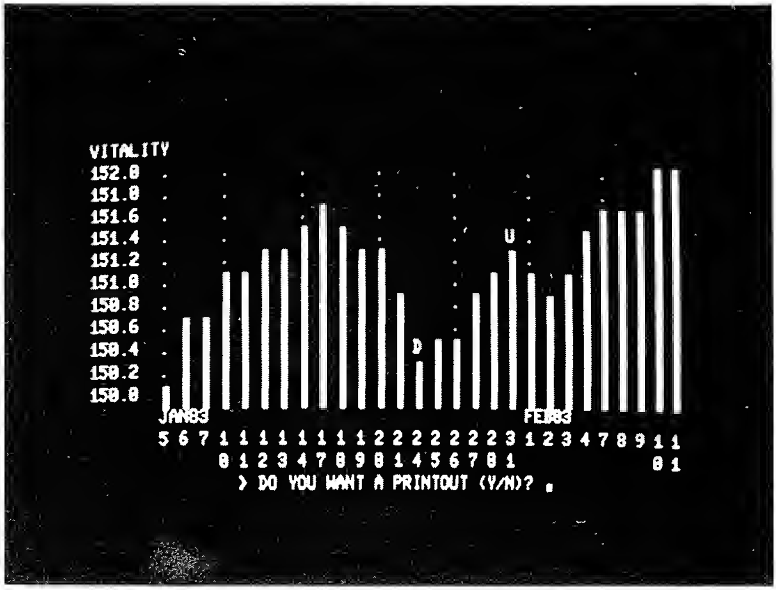
LINE 230: Removes the trailing blank from the unit number.

LINE 240: Centers each unit number directly over the appropriate bar.

LINE 250: Terminates the loop.

LINES 260-270: Waits for you to press a key, then loops back to input data for another chart.

In a sense, this program makes a diverting little game; it's fun to input numbers and watch the computer create bar charts based on them. But this program also represents an extremely useful utility. The routine in lines 100 through 250, which actually draws the chart, could be used in other programs.



Bar chart on a TRS-80

CUSTOMIZING THE PROGRAM

One useful application would be a program that reads data from a disk or cassette tape and creates a bar chart or charts based on that data. You would need, of course, to write a second program to place that data on the disk. The data for each chart would need to include the program name, the number of bars, the name of the standard unit, the number of units for each bar, and the length of the longest bar. Data read from the disk would need to be assigned to variables like this:

NB = Number of bars
T$ = Title
SU$ = Name of standard unit
L(0) . . . L(NB −1) = Length of bars 1 to NB
HB = Length of longest bar

The chart-drawing routine could then be called to create the chart. By placing a special separator symbol between

groups of data on the disk, you could have the program print out several charts in sequence, displaying each until a key is pressed, then drawing the next. Such a program would be ideal for business presentations (for example, graphing the amount you earn each week on a paper route, or mowing the lawn), or simply for impressing your friends.

You may also wish to make a few changes in the chart-drawing routine itself. For instance, it would be useful if the user could give a name to each bar, so that the significance of the individual bars would be apparent. If the bars represent sales figures for consecutive years, the first bar could be labeled 1979, the second bar 1980, and so forth. The name could be placed underneath the bar, where the bar numbers are placed in the current version. Of course, when a large number of very narrow bars are displayed, the name might not fit underneath each bar; hence, you might want to redesign the way the bars are displayed. One possible rearrangement would be for the bars to grow horizontally rather than vertically, with the individual bars stacked one over top of the other. The names of the bars could be placed in a vertical column on the right-hand side of the display. Of course, this would reduce the number of bars that could be shown, since there are fewer pixels vertically on the display.

If you have access to a plotter—a printer capable of reproducing graphics on paper—it would be valuable to add an option that would produce a hard copy of the chart, rendering the chart on paper with the same procedures (appropriately modified) that we have used here to draw the charts on the video display.

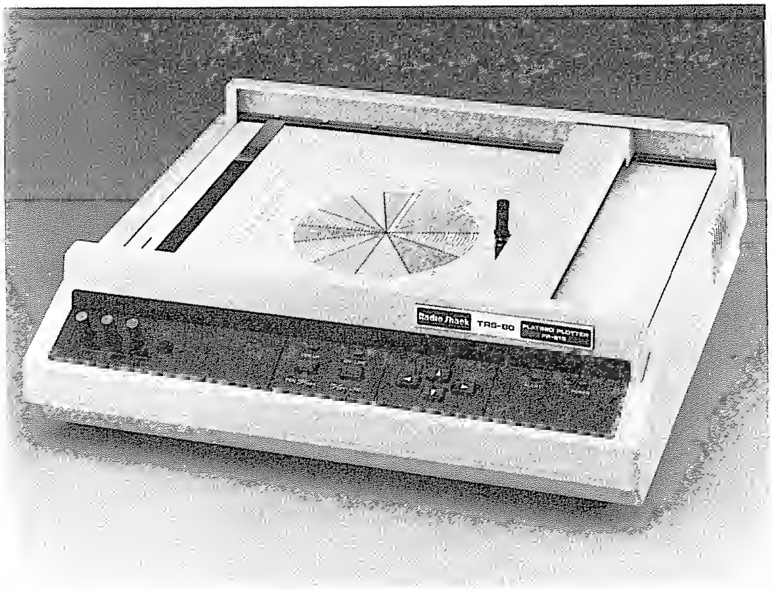
THE GRAPH PROGRAM

Bar charts are not the only visual form in which numeric information can be displayed. Useful, in addition, is the ability to graph data on a computer screen, that is, to display the data as a series of points, all of which are connected by lines.

Here, for instance, is a variation on our bar chart program that uses the line-drawing subroutine from Chapter Two to display a graph on the TRS-80 display:

```
10 DIM H(30)
20 CLS : MH = 0
30 INPUT "NUMBER OF POINTS";NP
40 FOR I = 0 TO NP -1
50 PRINT "HEIGHT OF POINT";I + 1; : INPUT H(I)
60 IF MH < H(I) THEN MH = H(I)
70 NEXT
80 CLS : X = 0 : FOR Y = 3 TO 46 : SET(X,Y) : NEXT : FOR X
= 0 TO 124 : SET(X,47) : NEXT
100 W = 121/(NP -1) : SU = 41/MH
110 FOR I = 1 TO NP -1 : SET(I * W,46) : NEXT
130 X1 = 0 : Y1 = 41-(H(0) * SU)
140 FOR I = 1 TO NP -1
150 X2 = W * I : Y2 = 41 -(H(I) * SU)
160 GOSUB 10000 : X1= X2 : Y1 = Y2
170 NEXT
180 K$ = INKEY$ : IF K$ = "" THEN 180
190 GOTO 20
10000 DX = X2 -X1 : DY = Y2 -Y1
10010 IF ABS(DX) < ABS(DY) THEN 10080
10020 YI = DY/ABS(DX)
10030 FOR X = X1 TO X2 STEP SGN(DX)
10040 SET(X,Y1)
10050 Y1 = Y1 + YI
10060 NEXT
10070 RETURN
10080 XI = DX/ABS(DY)
10090 FOR Y = Y1 TO Y2 STEP SGN(DY)
10100 SET(X1,Y)
10110 X1 = X1 + XI
10120 NEXT
10130 RETURN
```

RUN the program. Instead of being prompted for the number of bars, you are prompted for the number of points you wish to graph, and the height of each point, that is, the number of units from the bottom of the screen to the point. However, you may use any coordinate system you wish when inputting this value; you need not restrict yourself to the forty-eight-pixel, Y-coordinate limit of the TRS-80 screen. The program automatically converts your number to TRS-80 coordinates.



TRS-80 graphics plotter with pie chart

Once you have input a height for all points, the points will be drawn from left to right and connected with lines. The internal workings of this program are similar to those of the bar chart program; we will not explain in detail how it works. (Recall that the subroutine in line 10000 is the same subroutine introduced in Chapter Two for drawing lines between specified points on the screen.)

You may wish to dress up this program with a number of useful options or alter the way in which data is input. The program could be used, for instance, to graph a data trend over a period of time, or to chart the grades earned by your classmates in an exam.

Ambitious programmers may want to dream up still other ways of displaying information graphically on the TRS-80 display. The pie chart, for instance, is a popular method of illustrating the percentages of a given amount (of money, for instance, or supplies) apportioned for a specific purpose. To create a pie chart, you would need to draw a circle, then draw angled lines from the center of the circle to the perimeter to delineate the "slices" of the pie. You are

warned, however, that this is a fairly advanced programming project, and requires at least a nodding acquaintance with trigonometry.

Needless to say, these are only a few of the ways in which you can display information graphically, given the assistance of a powerful calculating and information processing engine like the TRS-80. If you can think of methods not suggested here, by all means try them out. If you should come up with a method that no one has thought of before, it may turn out to be your ticket to fame and fortune.

Or you might only discover that playing with computer graphics is a heck of a lot of fun.

And that's almost as nice.

INDEX

ABOUT THE AUTHOR

Christopher Lampton is the author of more than twenty books for Franklin Watts, including a number of popular First Book and Impact titles. Of late he has turned his attention to the world of computers, writing on a variety of programming languages and teaching the basics to beginners. He has written all the books on computer languages and graphics in Watts' Computer Literacy Skills series.

Chris first became a computer enthusiast when he purchased a Radio Shack computer to use for word processing. He has since acquired eight more computers.

Chris lives in Maryland, right outside Washington, D.C., and has a degree in broadcast journalism. In addition to his books in the areas of science and technology, he has written four science-fiction novels.

ABOUT THE AUTHOR

Christopher Lampton is the author of more than twenty books for Franklin Watts, including a number of popular First Book and Impact titles. He has written all the books on computer languages and graphics in Watts' Computer Literacy Skills series.

Chris first became a computer enthusiast when he purchased a Radio Shack computer to use for word processing. He now owns a total of eight computers.

Chris lives in Maryland, right outside Washington, D.C., and has a degree in radio and television broadcasting. In addition to his books in the area of science and technology, he has written four science-fiction novels.

PRINTED IN THE UNITED STATES OF
AMERICA BY MOFFA PRESS, INC.

FRANKLIN WATTS' COMPUTER LITERACY SKILLS SERIES

ADVANCED BASIC

BASIC FOR BEGINNERS

COBOL FOR BEGINNERS

FORTH FOR BEGINNERS

FORTRAN FOR BEGINNERS

GRAPHICS AND ANIMATION ON THE COMMODORE 64

GRAPHICS AND ANIMATION ON THE TRS-80

PASCAL FOR BEGINNERS

PILOT FOR BEGINNERS

6502 ASSEMBLY-LANGUAGE PROGRAMMING

Z80 ASSEMBLY-LANGUAGE PROGRAMMING

531-10059-6
4009